KU LEUVEN

Software Ingenieurstechnieken

3elicti

Examenvragen (2014-2015)

Auteur:
Gilles Callebaut

Prof.:
dr. ir. Annemie Vorstermans

7 juni 2015

Voor meer informatie of bij fouten, contacteer gilles.callebaut@student.kuleuven.be

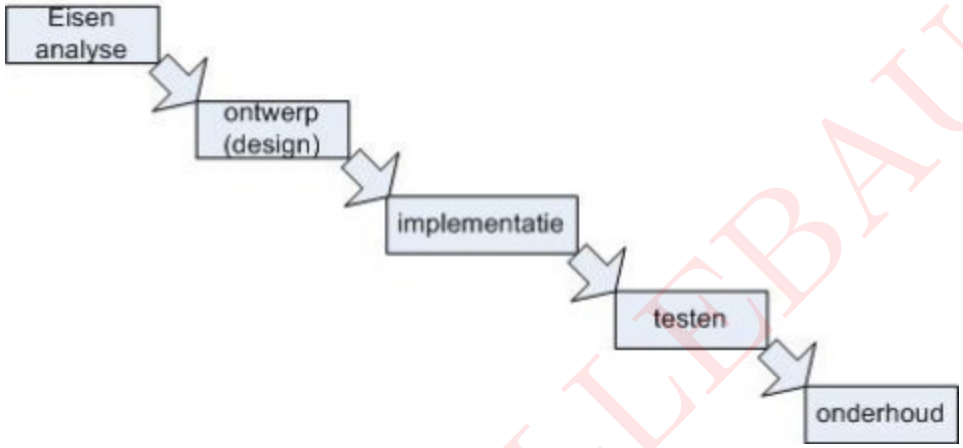
Inhoudsopgave

1 Wat is software engineering? Teken en leg uit: klassieke levenscyclus, prototyping en spiraalmodel

1.1 Definitie software engineering

Het instellen en gebruiken van gezonde ingenieursprincipes om economische verantwoorde software te verkrijgen dat betrouwbaar is en effeciënt werkt op reële machines.
Er wordt verwacht dat door het invoeren van een methodologie software met een betere kwaliteit kan geleverd worden. Voorbeelden van methodologiën zijn klassieke levenscyclus, prototyping, RUP, SCRUM, ...

1.2 Klassieke levenscyclus



Figuur 1: Klassieke levenscyclus (watervalmodel)

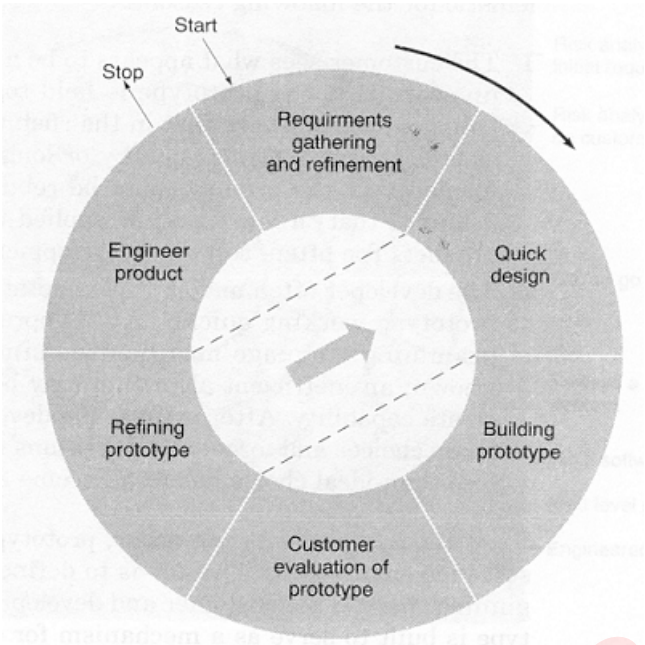
Het begrip "klassieke levenscyclus" heeft een weerslag op alle fasen die de software doorloopt vooraleer deze uit productie verdwijnt. Deze fasen worden samengevat in het watervalmodel (zie figuur 1).
Het watervalmodel is een lineaire opvatting van software development. Dit houdt in dat de software 5 fasen doorloopt:

- **Analyse**: Onderzoek en overleg over de vereisten en specificaties van de software. Hieronder valt ook het inschatten van de gevolgen van deze eisen.
- **Ontwerp**: Gedetailleerde uitwerking van de structuur en architectuur, de werking van de toekomstige software (UML) om dan uiteindelijk het ontwerp te kunnen omzetten in programmacode.
- **Implementatie**: Het eigenlijke programmeerwerk.
- **Testen**: De software dient uitvoerig getest te worden. Bij het watervalmodel kan er pas getest worden als de software al af is.
- **Onderhoud**: Al het werk dat wordt verricht na het in productie nemen van het eindproduct.

Het nadeel hiervan is dat we te laat testen. Er is een grote kans op een ontwerpfout waardoor we helemaal opnieuw moeten beginnen.

1.3 Prototyping

Prototyping is het maken van een vroegtijdige versie van de software voor demonstratiedoeleinden. Dit kadert meestal in de implementatiefase van een andere ontwerpmethodologie. Als de klant niet goed weet wat hij wilt dan wordt er snel een simpel ontwerp gemaakt. Dit ontwerp wordt uitgebouwd tot een prototype, een proefprogramma. Dit prototype wordt vervolgens naar de klant gestuurd voor evaluatie. Wanneer blijkt dat de klant niet tevreden is met het resultaat wordt een nieuw ontwerp gemaakt. In het andere geval wordt het prototype verder afgewerkt en uitgebracht.
Dit model wordt vaak toegepast voor UI en technologische nieuwigheden.



Figuur 2: Prototyping

1.4 Spiraalmodel

Dit is een **metamodel** dat in feite elk ander model kan bevatten (met uitzondering van het watervalmodel). Het spiraalmodel bestaat uit 4 stappen die steeds herhaald kunnen worden: Planning, Risico analyse, engineering: softwareontwikkeling en evaluatie door de klant.

Het model is gebaseerd op het uitvoeren van activiteiten die het hoogste risico met zich meebrengt. Men begint dus met het uitvoeren van het onderdeel met het hoogste risico. Op die manier wordt het mogelijk gemaakt om een bepaald deel van het systeem te implementeren, terwijl een ander deel nog ontworpen moet worden.

Mocht een bepaald onderdeel te grote risico's met zich meebrengen dan kan er voor worden gekozen dit onderdeel niet uit te voeren, of het project in zijn geheel af te blazen. Door de volgorde van werken zou dit aan het begin van het project moeten blijken.

1.4.1 De vier fasen

- Bepaal het doel, de alternatieven en de beperkingen.
- Tijdens de tweede fase wordt er gekozen om te beginnen met het onderdeel dat de meeste risico's bevat (met hoge moeilijkheidsgraad).
- In het overgangsgebied wordt er beslist of er wordt verder gewerkt ,**'GO or NO GO'**.
- Ontwikkel het 'volgend-niveau-product'
- Voer een review uit en plan de volgende doorgang van deze vier fasen. Beëindig dit project, of begin weer bij stap 1.

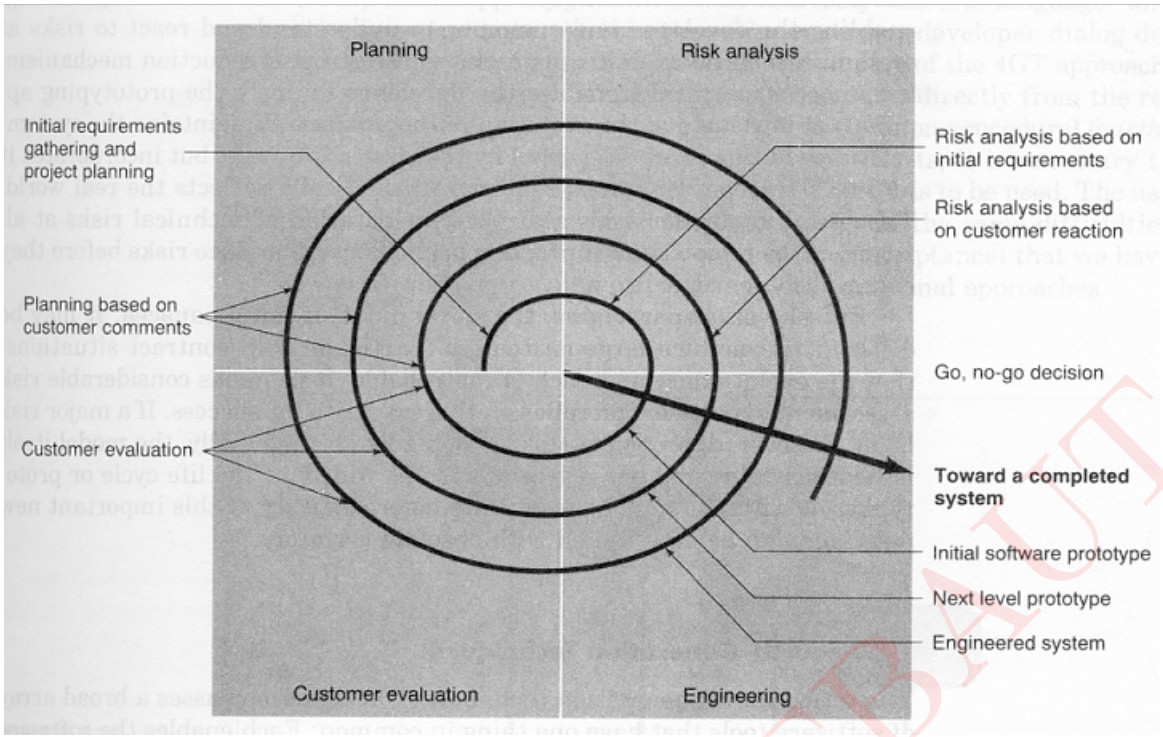
1.4.2 Voordelen

De meest risicodragende problemen, worden in het begin van een project ingeschat en onder handen genomen. Het vertrouwen in het slagen van het project is hierdoor groot. Ook wordt eerder duidelijk dat een project onuitvoerbaar blijkt te zijn.

Risico's zijn voor een opdrachtgever kleiner, dan bij gebruik van de watervalmethode.

1.5 Nadelen van deze methodologiën

Het maken van een planning is vaak onmogelijk, omdat het van tevoren inschatten van de risico's, of iteraties niet voldoende kan. De opdrachtgever kan er daardoor niet zeker van zijn, wanneer het project

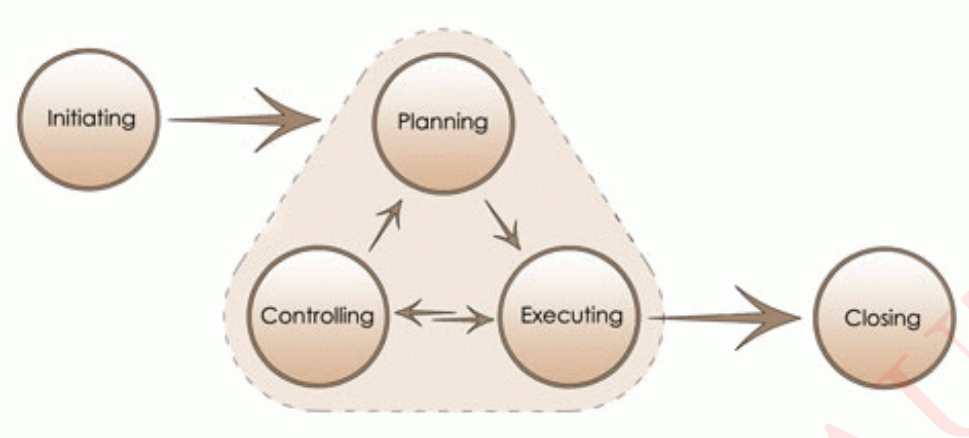


Figuur 3: Spiraalmodel

zal zijn afgerond. Dit is overigens voor het watervalmodel ook van kracht. Daar is het onzeker wat de doorlooptijd zal zijn van de verschillende fasen.

Het inschatten van risico's is een centraal thema. Dit is vaak een moeilijke taak, waardoor het nodig is iemand aan te stellen die raad weet met complexe, risicovolle en grote projecten. Daardoor is het project afhankelijk van de kwaliteit van mensen en methoden voor het vaststellen van de risico's.

2 Welke fasen kan men bij projectmanagement onderscheiden? Bespreek ze kort.



Figuur 4: 5 fasen bij projectmanagement

Men, kan 5 fasen onderscheiden bij projectmanagement:

- Initiatiefase
- Planningsfase
- Uitvoeringsfase
- Controlefase
- Afsluitfase

2.1 Initiatie

2.1.1 Project aanname

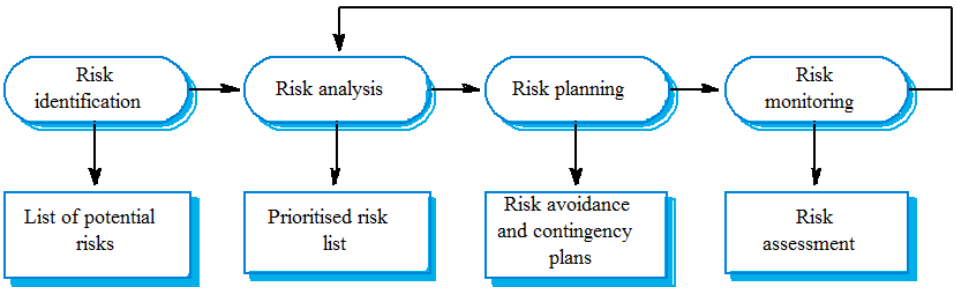
In deze fase zal men beslissen of men het project al dan niet zal aannemen. Deze beslissing zal gebeuren rekening houdend met verschillende factoren. Het project kan moreel of praktisch onmogelijk (niet genoeg kennis, mensen, tijd) zijn, men kan er voor kiezen het project uit te stellen naar een latere datum omdat andere projecten een hogere prioriteit hebben. Ook zal onderzocht worden of het eindproduct wel een goede marktpositie zal verzekeren.

2.1.2 Doelstellingen van het project definiëren

Het doel van het project kan zijn om een duidelijk voordeel te leveren om zo de klant kosten en tijd te besparen en uiteindelijk meer winst te maken voor de onderneming. De verwachte resultaten en winsten moeten duidelijk beschreven worden. Er moeten criteria vastgesteld worden om het eindresultaat te beoordelen (wanneer is het project geslaagd? i.f.v. tijd, kost, resultaten). Iedereen moet akkoord gaan hiermee (consensus). Men moet vermijden dat men een goed product maakt dat mensen niet willen gebruiken. Goede criteria zijn specifiek en realistisch. Ook moet er een einddatum vastgesteld worden die haalbaar is, te korte einddatum resulteert (bijna) altijd in mislukking. De doelstellingen moeten meetbaar zijn en de kwaliteit uitdrukbaar zijn.

2.1.3 Risico's en beperkingen

Het project moet doenbaar zijn binnen de beperkingen van de huidige economie. Er moet ook altijd rekening gehouden worden met de politiek, wetten en structuren. Er moet ook een inschatting gemaakt worden van mogelijke risico's zoals weersfactoren, elektriciteitspannes, stakingen, leveranciers die failliet gaan, ... Ook budgetten en het aantal beschikbare mensen zijn beperkt. Ten laatste bestaan er ook technologische risico's: heeft de huidige technologie wel genoeg mogelijkheden om te bereiken wat we willen? We moeten dus doen aan risk management. Dit houdt in dat we risico's moeten identificeren en



Figuur 5: Risk management

plannen moeten opstellen om het effect op het project te minimaliseren. Het risk management proces gaat als volgt:

- Risk identification: identificeren van project, product en business risico's
- Risk analysis: beoordelen waarschijnlijkheid en gevolgen van deze risico's
- Risk planning: opstellen plannen om effecten te vermijden of te minimaliseren
- Risk monitoring: houd risico's in de gaten tijdens hele verloop project, dit wordt echter pas gedaan in de controle-fase.

2.1.4 Bouwen van een team

Mensen hebben een verschillende persoonlijkheid, kennis en vaardigheden. Bij het verdelen van de taken van het project zullen we dus moeten kijken naar het niveau van kennis: expert of basiskennis, communicatieve vaardigheden, ... Hoeveel supervisie is er nodig, moeten we een deel van het project uitbesteden aan bijvoorbeeld consultants.

2.2 Planning

In de planningsfase verfijnen we het bereik van het project. Er moet een evenwicht bestaan tussen resultaten, tijd en middelen. We moeten een takenlijst opstellen om het doel te bereiken. Het grote project moet verdeeld worden in kleinere beheersbare taken. Deze taken moeten dan uitgevoerd worden in de meest efficiënte sequentie. Voor elke taak moet een schatting gemaakt worden van de benodigde tijd. Nadien moeten we een rooster maken en de beschikbare middelen (mensen) aan de verschillende taken toekennen.

2.2.1 Opsplitsen project in beheersbare taken

Dit gebeurt door de meest logische volgorde van taken te identificeren. Te kijken of er interferentie is van taken en dan het nodige werk te beoordelen en controleren. Wat is de vereiste kennis en het nodige aantal mensen voor elke taak.

2.2.2 Wat is een taak?

Een taak is duidelijk, eenduidig (geen verwarring mogelijk) en voldoende gedetailleerd. Een taak moet 1 geheel vormen, ook in de tijd. De taken moeten kunnen gemeten worden en in het schema kunnen gevoegd worden.

2.2.3 Mijlpaal

Een mijlpaal wordt gezet wanneer een belangrijke verzameling van taken (subproject) is afgehandeld. Deze mijlpalen moeten meetbaar zijn en getoond worden via netwerkdiagrammen.

2.2.4 Netwerkdiagramma

Een netwerkdiagram toont de sequenties en relaties tussen taken en kan mijlpalen identificeren (voor controle van het verloop). Deze is opgesteld vanuit de taaklijst en heeft een grafische voorstelling.

2.2.5 Kosten

We moeten ook rekening houden met de loonkosten, noden (toestellen, software, papier, telefoon), inhuren experten, ...

2.3 Uitvoering

Tijdens de uitvoeringsfase moet het team geleid worden. Er moet vergaderd worden met de teamleden, en er moet communicatie zijn tussen het management en de klanten. Enkel zo kunnen we problemen en conflicten oplossen en kunnen we op voorhand zorgen voor voldoende middelen.

2.3.1 Projectbijeenkomst

In de projectbijeenkomst is het de bedoeling om

- de doelen van het project mee te delen
- zorgen dat iedereen zijn eigen objectieven en verantwoordelijkheden kent
- zorgen dat iedereen gemotiveerd en enthousiast is
- duidelijk maken wie leider is en zorgen dat iedereen jou wil volgen
- identificeren van kritische deadlines en fasen van project
- uitleggen van procedures (rapporten, vergaderingen)
- expliciet de leden verantwoordelijkheden geven voor de initiële taken

Dit doen we door een welgevormde en duidelijke presentatie te geven, plus achteraf feest (en dat liefst ook bij elke belangrijke mijlpaal)!

2.3.2 Leiding

Een goede leider luistert naar zijn teamleden, stelt veel vragen, observeert wat er gebeurt en wil graag veel weten over dingen die hij niet weet (of onvoldoende). Men moet ook beschikbaar zijn, beslissingen durven nemen, werk delegeren, maar let op dat je niet over-managed, laat de leden hun eigen werk doen.

2.3.3 Communicatie

Er moet regelmatig groepsvergaderingen georganiseerd worden die kort en to-the-point zijn en die vooral **niet** mogen **afgelast** worden! Er moet ook mogelijkheid zijn tot individuele gesprekken, waarbij alles bespreekbaar is. De leden mogen geen schrik hebben om feedback te geven over de projectstatus. Dit gaat best door informele gesprekken met een koffie of tijdens het eten. De communicatie wordt best zo weinig mogelijk gedaan onder de vorm van geschreven boodschappen, dit omdat we niet willen dat berichten gelezen worden door onbevoegden. Het is ook belangrijk om de reacties van de persoon te zien tijdens de communicatie.

2.4 Controle

In de controle fase moet er worden gekeken of we niet afwijken van het plan, desnoods moeten we het plan corrigeren. In deze fase behandelen we ook de aanvragen voor projectwijzigingen en herschikken de middelen als nodig. Het kan ook zijn dat we in deze fase terug moeten gaan naar planning. Opgelet bij aanpassingen, communiceer en zorg voor de nodige goedkeuringen.

2.4.1 Succescriteria voor monitoring

- vergelijk projectverloop met plan
- plan (schema, budget) aanpassen om project op goede spoor te houden
- aanpassingen: toepasselijk en goedgekeurd
- documenteer vooruitgang en wijzigingen
- wees betrokken: niet in ivoren toren wachten op de resultaten

2.4.2 Conflicten

- niet achter hetzelfde doel staan, omdat er een misverstand is van de verwachte resultaten of door het niet respecteren van de volgorde van de taken
- taakbeschrijving niet specifiek genoeg
- administratieve procedures (aantal, verspreiding "geheime" informatie)
- teamleden weten niet goed wat ze moeten doen
- technische onzekerheid (ontwikkeling nieuwe technologie)
- verdeling van taken, bv saaie taken vs leiden subgroep
- budgetten, bv deelgroep verbruikt teveel
- schema: niet genoeg tijd om taak te volbrengen
- persoonlijke conflicten

De meeste problemen komen vaak door geen of een te late start, niet genoeg tijd, teveel rapporteren en te weinig communicatie, wegmoffelen van achterstand, ...

2.5 Afsluiten

Wat hebben we bereikt, wat hebben we geleerd van deze ervaring? In deze fase is het de bedoeling dat het projectverloop en - resultaten worden besproken met de teamleden en het management, alsook het maken van een rapport.

3 Wat zijn de gouden regels voor een projectleider

3.1 Zoeken naar consensus voor projectresultaten

We moeten goed weten wat we moeten bereiken. Er moet een consensus bestaan tussen management, team en klant.

3.2 Zoeken naar het beste team

We moeten een team zoeken die gewillig is en een voldoende kennis heeft voor het project. We moeten de mensen motiveren en het werk verdelen. Ook al hebben ze niet genoeg ervaring of training.

3.3 Ontwikkeling plan

We moeten een goed plan ontwikkelen die we up-to-date moeten houden. Een goed plan is

- Compleet
- Gedetailleerd
- Toepasselijk

3.4 Hoeveel middelen

We moeten bepalen hoeveel middelen we echt nodig hebben voor het project. Als er middelen niet beschikbaar zijn dan onderhandelen we wat er uitgevoerd zal worden.

3.5 Zorg voor realistisch schema

Vermits we geen tijd kunnen bijmaken moeten we een realistisch schema opbouwen, waarbij er niet te veel tijdsdruk is.

3.6 Niet meer doen dan nodig

Zorg ervoor dat je niet meer doet dan nodig zodat wat je doet duidelijk is voor iedereen.

3.7 Succes afhankelijk van mensen

Onthoud dat mensen tellen, het succes van een project hangt sterk af van mensen.

3.8 Ondersteuning van management winnen

Zorg ervoor dat je zowel formele als blijvende ondersteuning van het management kan winnen. Dit vraagt echter communicatieve en onderhandelingsvaardigheden.

3.9 Aanpassingen (durven) voeren

- Onvoorziene omstandigheden: computerpanne, waterschade
- Nieuwe informatie, nieuwe tool
- Nieuwe eisen van de klant

3.10 Mensen informeren

"If they know nothing of what you are doing, they suspect you are doing nothing." (Graham's Law)

3.11 Nieuwe zaken uitproberen

Je moet bereid zijn om nieuwe zaken uit te proberen. Vermits alle projecten verschillen waardoor je niet te star moet vasthouden aan 1 formule.

3.12 Word een leider

- Eenvoudig voor sommigen, hard werk voor anderen
- Een leider moet plannen, opvolgen en controleren
- Hij is bron van wijsheid en is de ondersteuning van mensen
- Een leider is geen dictator, maar een lid van het team

4 Wat zijn taken en hoe kaderen zij zich binnen een project?

In de planningsfase verfijnen we het bereik van het project. Er moet een evenwicht bestaan tussen resultaten, tijd en middelen. We moeten een takenlijst opstellen om het doel te bereiken. Het grote project moet verdeeld worden in kleinere beheersbare taken. Deze taken moeten dan uitgevoerd worden in de meest efficiënte sequentie. Voor elke taak moet een schatting gemaakt worden van de benodigde tijd. Nadien moeten we een rooster maken en de beschikbare middelen (mensen) aan de verschillende taken toekennen.

4.1 Opsplitsen project in beheersbare taken

Dit gebeurt door de meest logische volgorde van taken te identificeren. Te kijken of er interferentie is van taken en dan het nodige werk te beoordelen en controleren. Wat is de vereiste kennis en het nodige aantal mensen voor elke taak.

4.2 Wat is een taak?

Een taak is duidelijk, eenduidig (geen verwarring mogelijk) en voldoende gedetailleerd. Een taak moet 1 geheel vormen, ook in de tijd. De taken moeten kunnen gemeten worden en in het schema kunnen gevoegd worden.

4.2.1 Schatten

We moeten de tijd voor elke taak kunnen schatten. De schatting is gebaseerd op: schatting van de mensen die taak zullen uitvoeren, objectieve expert, analoge taken in vroegere projecten. We kunnen echter gebruik maken van LOC (lines of code), FP (function points) en COCOMO II om een project duur in te schatten.

LOC Het probleem bij LOC is dat de broncode een klein onderdeel is van de totale ontwikkeling, het is programmeertaal afhankelijk, telt commentaar en declaraties mee?, hebben we herbruikbare code gebruikt, is een hogere LOC gelijk aan betere code?, is er gebruik gemaakt van codegeneratoren of andere tools? ....

FP *Function points* zijn gebaseerd op een combinatie van programma karakteristieken, externe inputs en outputs, gebruikersinteracties, externe interfaces, bestanden die gebruikt worden door het systeem, ... Met elke karakteristiek wordt er een gewicht geassocieerd, de FP is dan gewoon de som van alle *raw counts* maal hun gewicht. De *raw count* is een getal dat de complexiteit van dat deel van het project voorstelt.

COCOMO (II) Het COCOMO (II) model is opgedeeld in sub-modellen:

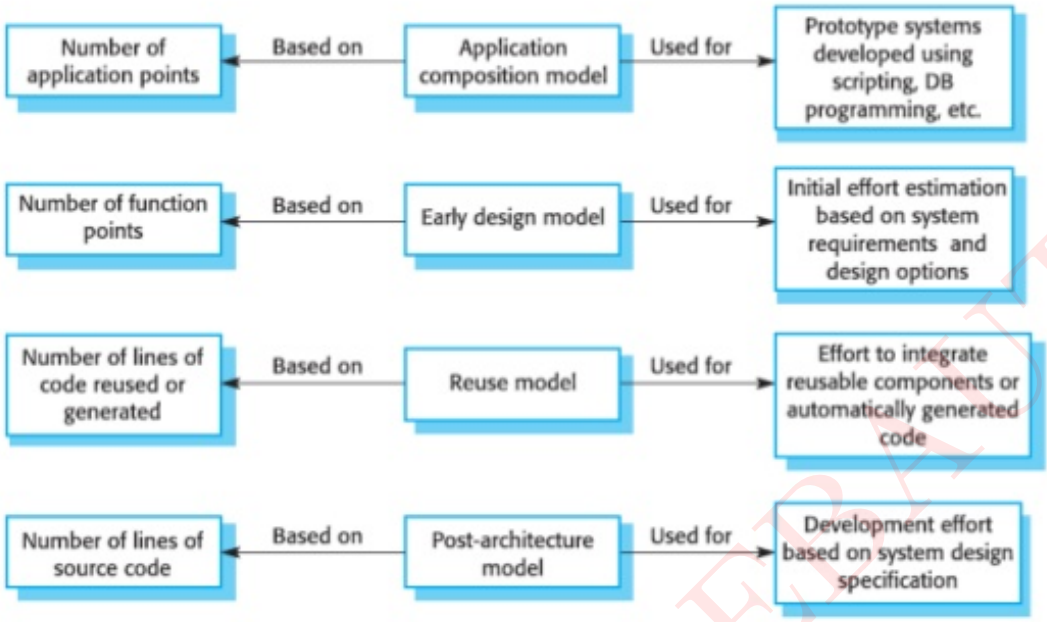
- Application composition model. Used when software is composed from existing parts.
- Early design model. Used when requirements are available but design has not yet started.
- Reuse model. Used to compute the effort of integrating reusable components.
- Post-architecture model. Used once the system architecture has been designed and more information about the system is available.

4.3 Mijlpaal

Een mijlpaal wordt gezet wanneer een belangrijke verzameling van taken (subproject) is afgehandeld. Deze mijlpalen moeten meetbaar zijn en getoond worden via netwerkdiagrammen.

4.4 Netwerkdiagramma

Een netwerkdiagram toont de sequenties en relaties tussen taken en kan mijlpalen identificeren (voor controle van het verloop). Deze is opgesteld vanuit de taaklijst en heeft een grafische voorstelling. In de grafische voorstellingen zien we de afhankelijkheden tussen taken en welke parallel lopen.



Figuur 6: Sub-modellen COCOMO II

ID	Task Name	Start	Finish	Duration
1	Task 1	7/02/2006	14/02/2006	1,2w
2	Task 2	14/02/2006	17/02/2006	,8w
3	Task 3	13/02/2006	22/02/2006	1,6w
4	Task 4	17/02/2006	27/02/2006	1,4w
5	Task 5	27/02/2006	20/03/2006	3,2w
6	Task 6	27/02/2006	10/03/2006	2w
7	Task 7	7/02/2006	24/02/2006	2,8w
8	Task 8	28/03/2006	7/04/2006	1,8w
9	Task 9	20/03/2006	14/04/2006	4w
10	Task 10	31/03/2006	17/04/2006	2,4w

Figuur 7: Grantt Chart voorbeeld

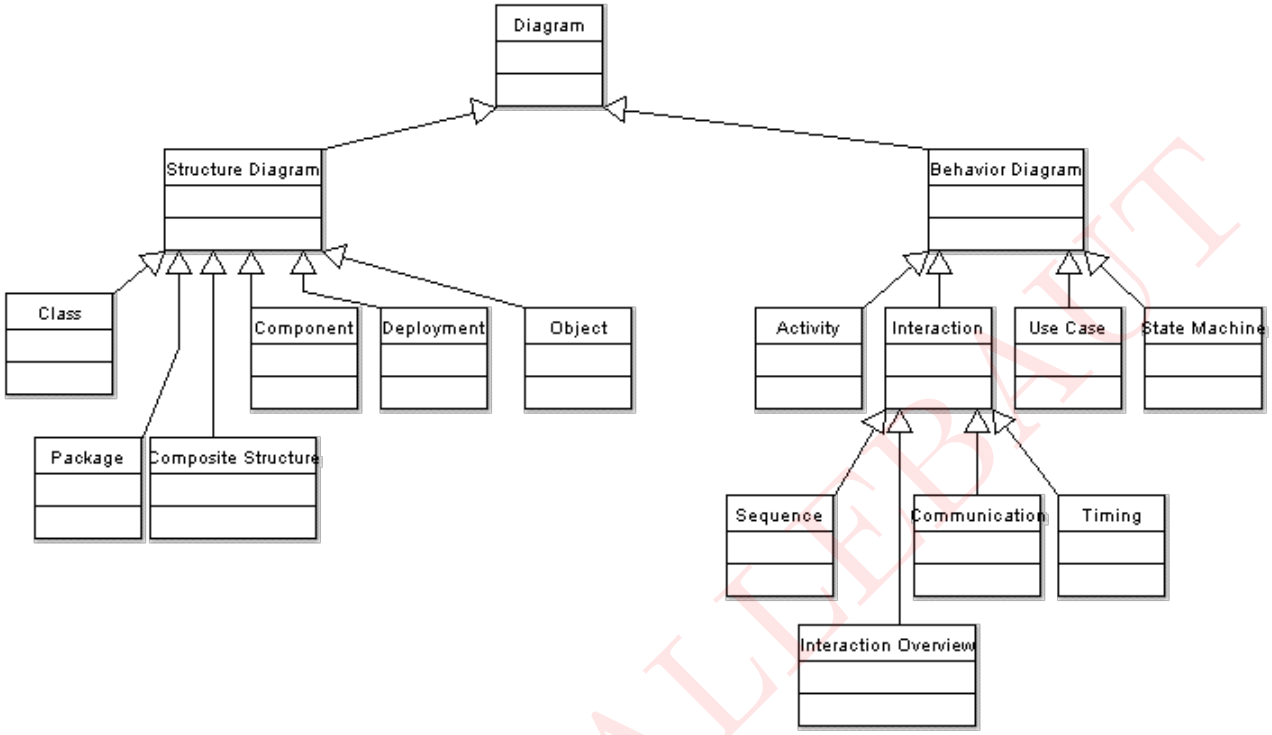
5 Welke conflicten kunnen tijdens het verloop van het project opduiken?

- niet achter hetzelfde doel staan, omdat er een misverstand is van de verwachte resultaten of door het niet respecteren van de volgorde van de taken
- taakbeschrijving niet specifiek genoeg
- administratieve procedures (aantal, verspreiding "geheime" informatie)
- teamleden weten niet goed wat ze moeten doen
- technische onzekerheid (ontwikkeling nieuwe technologie)
- verdeling van taken, bv saaie taken vs leiden subgroep
- budgetten, bv deelgroep verbruikt teveel
- schema: niet genoeg tijd om taak te volbrengen
- persoonlijke conflicten

De meeste problemen komen vaak door geen of een te late start, niet genoeg tijd, teveel rapporteren en te weinig communicatie, wegmoffelen van achterstand, ...

6 Wat is UML? Geef ook een lijst van de diagrammen met een korte uitleg.

De Unified Modeling Language, afgekort UML, is een modelleer taal om objectgeoriënteerde analyses en ontwerpen voor een informatiesysteem te kunnen voorstellen. UML is een verzameling van visuele modelleertalen. UML is een standaard ontwikkeld door Object Management Group (OMG).



Figuur 8: UML-diagrammen

Voor een meer gedetailleerde uitleg, zie mijn samenvatting van UML.

6.1 Structuurdiagrammen

- Class Structuur van de applicatie, opbouw in klassen
- Composite structure
  Interne structuur (*in parts*) van klasse van een component
- Component
  Fysieke componenten en afhankelijkheden (architectuur). Welke files, libraries, ... worden er gebruikt.
- Deployment
  Run-time configuratie van applicatie. Welke delen draaien op welke toestellen en hoe verloopt de communicatie ertussen.
- Object
  Relatie tussen objecten. Deze diagrammen tonen voorbeeldobjecten die mulipliciteiten, reflectieve associaties, ... illustreren.
- Package
  Afhankelijkheden tussen packages, groeperen van modelelementen. Dit wordt echter enkel gedaan als de modellering op één pagina moet worden voorgesteld.

6.2 Gedragsdiagrammen

- Activity
  Processen moddeleren, het is ook het enige diagram dat parallele zaken kan voorstellen.

- Use Case
  functionele eisen vastleggen door het systeem als een zwarte doos voor te stellen. Dit is trouwens ook het eerste diagram dat men opstelt.
- State Machine
  Levenscyclus (inclusief de toestanden en overgangen) van 1 object weergeven.
- Interaction
  - Sequence
    Samenwerking tussen objecten, waarbij de nadruk ligt op tijd. De volgorde van de interacties worden per use case weergegeven.
  - Communication
    Samenwerking tussen objecten, waarbij de nadruk ligt op ruimte. De interacties die doorgaan worden per use case weergegeven.
  - Interaction Overview
    Dit is een soort activiteitendiagram dat interactie modelleert over meerdere use cases.
  - Timing
    Interactie tussen objecten in de tijd. Dit is vooral belangrijk bij real-time systemen.

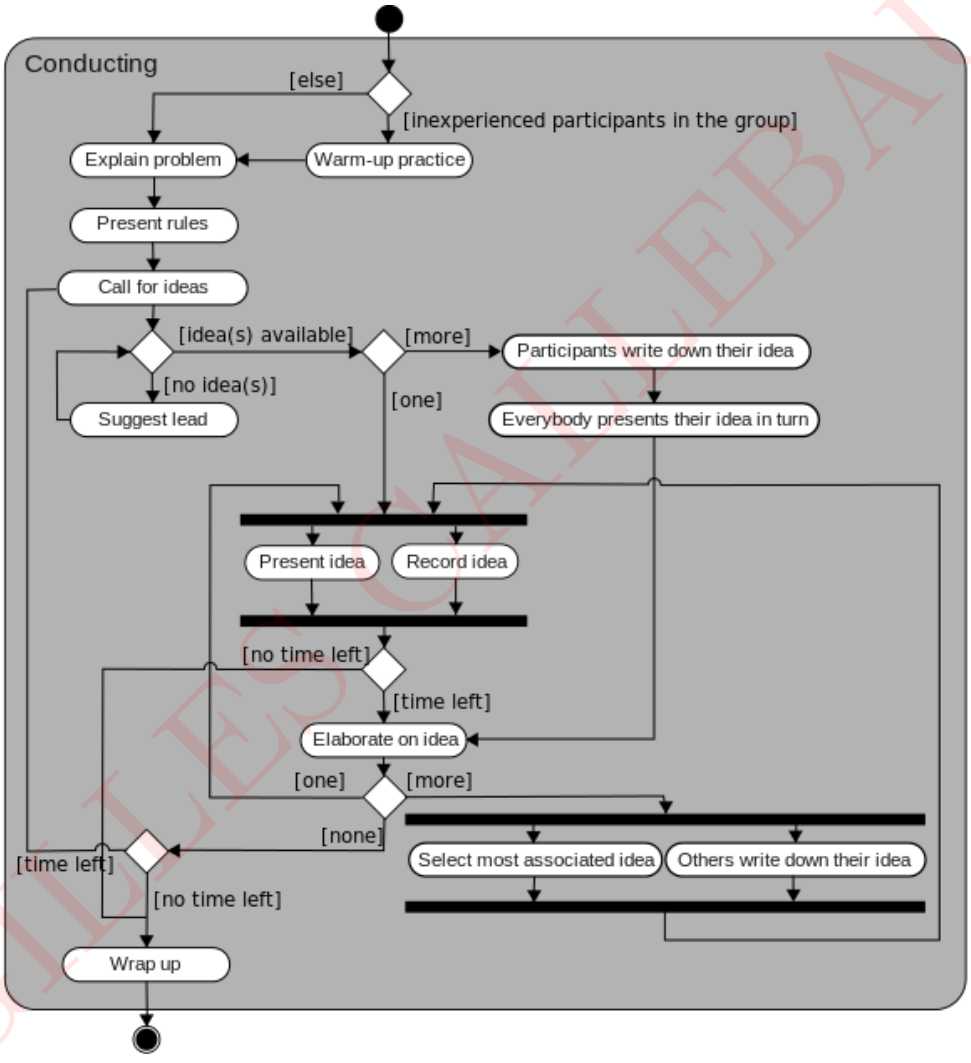
7 Dynamisch gedrag kan men modelleren met interactie-, toestands- en activiteitendiagrammen. Wanneer gaat men wat gebruiken? Geef een voorbeeld van elk.

7.1 Activiteiten diagram

Activiteiten diagrammen worden gebruikt om de workflow te modelleren van één use case. Een activiteiten diagram wordt gebruikt voor het inwendige van één methode voor te stellen/modelleren. Wat willen we **niet** zien in een activiteiten diagram.

- Hoe objecten samenwerken → Interactiediagrammen
- Hoe objecten zich gedragen → Toestandsdiagrammen

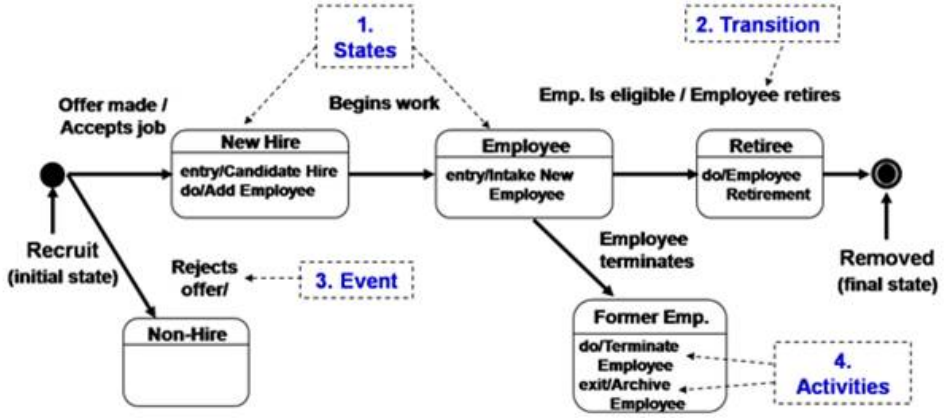
Een voorbeeld is een brainstorm proces, dit is weergegeven in figuur 9. Opmerking bij de figuur, er moet een *merge point* gebruikt worden voor *wrap up*.



Figuur 9: Activiteitendiagram van een brainstorm proces

7.2 Toestandsdiagram

Een toestandsdiagram wordt gebruikt om het gedrag van één object te beschrijven. Hierbij worden alle toestanden besproken en de overgang tussen die toestanden (via events). Het is dus een moderne versie van de *statechart*.
Voorbeeld: Bij gebruik van UI of zoals weergegeven op figuur 10 op de volgende pagina, de toestand van een persoon van sollicitant tot z/haar pensioen.



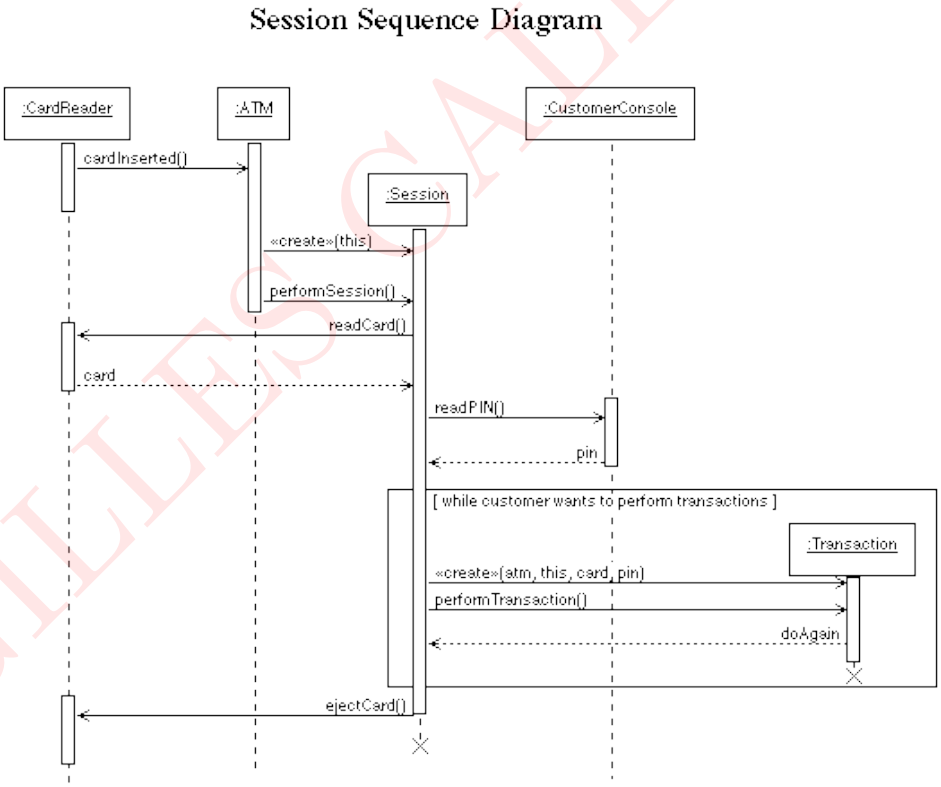
Figuur 10: Toestandsdiagram van een solicitatie

7.3 Interactiediagram

Deze diagrammen beschrijven hoe groepen van objecten samenwerken die typisch het gedrag van één use case vertegenwoordigen. Wat willen we **niet** zien in een interactie diagram.

- Gedrag van één object over meerdere use cases → Toestandsdiagrammen
- Gedrag van meerdere use cases → Interactieoverview

Een voorbeeld van een interactiediagram is een transactie in een ATM (zie figuur 11). Voor een meer



Figuur 11: Interactiediagram van een ATM transactie

gedetailleerde uitleg, zie mijn samenvatting van UML.

8 Waarmee moet men rekening houden bij het ontwerp van een *User Interface*?

- De eerste iteraties moeten zo goedkoop mogelijk zijn, d.m.v. schets, *mock-up*[1] en wireframes.
- Usability
  - Learnability, hoe gemakkelijk is het aan te leren?
  - Efficiency, hoe snel is het te gebruiken?
  - Memorability, hoe gemakkelijk is het te onthouden?
  - Errors, beperkt in aantal en herstelbaar
  - Satisfaction, tevredenheid, aangenaam om te gebruiken?
- Menselijke capaciteiten
  - Bewegende beelden, 10 frames/seconde (liefst 20)
  - Responsie
    Reponsietijd < 100 ms $\rightarrow$ onmiddellijk
    Als twee events langer dan 100ms uit elkaar liggen lijkt het alsof het ene het gevolg is van het andere. Een responsie moet wel zichtbaar zijn.
  - Motoriek
    Aanwijstaken zijn moeilijk en vragen tijd.
  - Geheugen
    Zorg voor connecties en associaties en niet enkel herhaling als de gebruiker iets moet aangeleerd krijgen.
    Zorg dat de informatie in *chuncks* worden voorgesteld en dat patronen kunnen herkend worden.
  - Kleur
    Geen onderscheid maken via kleuren zoals groen en rood of blauw en geel. Zorg er ook voor dat je nooit blauw en rood op elkaar zet.
- Nielsen's 10 principes
  1. Match the real world
     Vermijd technische jargon en toon de taal van de gebruiker.
     Ondersteun synoniemen bij zoekacties, gebruik echter zelf geen synoniemen in het programma.
  2. Consistency & Standards
     Geef gelijke dingen een analoog uitzicht en gedrag en het tegengestelde voor verschillende dingen.
     Let wel op, interne consistentie binnen de applicatie PLUS externe consistentie met andere applicaties op hetzelfde platform.
  3. Help & Documentatie
     Gebruikers lezen geen manuals, maar toch moeten er manuals en online help ter beschikking zijn. Tevens moeten deze doorzoekbaar, taak georiënteerd, kort en concreet zijn.
  4. User control & Freedom
     Gebruikers mogen niet bang zijn om in de val (vast) te lopen. We voorzien dus cancel-knoppen, undo-knoppen en laten lange operaties onderbreekbaar zijn.
  5. Visibility of system status
     Geef de gebruiker feedback via verandering van de cursor, selectie te *highlighten*, statussen te tonen, ...
  6. Flexibility & Efficiency
     Voorzie veelvuldig gebruik van shortcuts.

[1] In de software-industrie komt het begrip tevens voor om vroeg in het ontwikkelproces het software-ontwerp qua gebruikersinterface te testen. Een mock-up wordt voornamelijk gebruikt voor demonstraties, lessen, evaluaties of promotie. Een mock-up krijgt pas de term prototype als het ontwerp ook echt werkt. Meestal zijn de mock-ups dus voorbeelden qua uiterlijk

7. Error prevention
   Zorg ervoor dat de gebruikers moeten selecteren boven typen, schakel illegale commando's uit en vermijd verschillende modes. Als je al modes voorzie, zorg dat ze maar tijdelijk zijn (shift, drag-and-drop,...).
8. Recognition not recall
   Zorg dat gebruikers minder moeten onthouden en werk daarom met menu's boven commando's, comboboxen boven text boxen, ...
9. Error reporting, Diagnosis & Recovery
   Wees precies in de foutboodschappen (in de taal van de gebruiker). Wees beleefd en nooit beschuldigend, het is altijd de fout van het systeem, nooit van de gebruiker. Wees constructief in de help-methode en verberg technische details tot erom gevraagd wordt.
10. Aesthetic & Minimalistic design *Less is more*, laat alles weg wat niet nodig is. Zorg voor weinig maar goed-gekozen kleuren en font, aligneer, ...

- User centric design
  Wie zijn de gebruikers? Wat moet elke gebruiker kunnen doen?
  - User analyse
    Wat zijn de kenmerken van de gebruiker (leeftijd, geslacht, opleiding, vaardigheden, cultuur, links- of rechtshandig, soort gebruiker ...)?

- Taak analyse
  Wat moet er kunnen, wat zijn de precondities (voorafgaande taken, informatie), wat zijn de stappen, resource- of tijdslimieten, waar wordt mijn taak uitgevoerd, ...
  De gevaren bij taak-analyse is dat er soms geen rekening wordt gehouden met observaties waardoor je niet enkel ziet wat de gebruikers doen maar ook waarom.

9 Geef 15 puntjes over het schrijven van ononderhoudbare code

1. Lieg in de commentaren. Je hoeft niet actief te liegen, zorg er gewoon voor dat de commentaren niet up-to-date zijn met de code.
2. Peper de code met commentaar zoals /* tel bij i 1 op */. Logische dingen uitleggen.
3. Zorg ervoor dat elke methode iets meer (of minder) doet dan de naam suggereert. Eenvoudig voorbeeld: een methode genaamd isValid(x) zou als bijwerking x naar een binair getal moeten converteren en opslaan in een database.
4. Het is efficiënter om stukken code te copy-pasten dan om kleine herbruikbare modules te programmeren.
5. Zet nooit commentaar bij een variabele. Details zoals hoe de variabele gebruikt wordt, zijn grenzen, zijn legale waarden, zijn niet van belang.
6. Probeer zoveel mogelijk op één lijn te plakken. Dit zorgt voor minder overhead, minder tijdelijke variabelen en zorgt ervoor dat de source file korter wordt.
7. Omsluit een if/else nooit met "{}", zij zijn syntactisch overbodig.
8. Gebruik zo lang mogelijke namen voor variabelen of klassen die in slechts één karakter van elkaar verschillen, liefst van zelfde type.
9. Gebruik de kleine letter 'l' (i.p.v. 'L') om aan te geven dat het over een long constante gaat.
10. Gebruik nooit de letter i als de binnenste lusvariabele. Gebruik "i" voor alles behalve integers.
11. Bij het benoemen van functies maak je best gebruik van abstracte woorden zoals do, functie, routine, ...
12. In Java is het mogelijk om methoden dezelfde naam als je klasse te geven, zonder dat dit constructoren zijn. Misbruik dit.
13. Maak zoveel mogelijk variabelen static.
14. Hou zoveel mogelijk out-dated methoden en variabelen in je code.
15. In C++ moet je zoveel mogelijk library functies overloaden met # define. Zo lijkt het alsof je een bekende library functie gebruikt, maar in werkelijkheid doe je iets helemaal anders.
16. C++ operator overloading misbruiken voor totaal andere bewerking dan operator zegt (verschil i++ en i+=1, als i++ i*2 doet)
17. Main 1 grote try-catch
18. Exception teruggeven uit methode
19. Variabelen met namen in andere talen
20. Variabelen met namen van afkortingen of zonder klinkers maken
21. Geen SI eenheden gebruiken maar converteren
22. Al vroeger gebruikte variabele opnieuw gebruiken
23. Ingegeven data niet controleren

Het best dat je de andere voorbeelden uit "How To Write Unmaintainable Code" van Roedy Green ook eens bekijkt.

10 Wat is white en black box testen. Hoe kan je modules integreren (geef ook voor- en nadelen, wat heb je ervoor nodig)?

10.1 Black box testen

Bij black box testen zal het programma worden aanschouwd als een zwarte doos. Een juiste invoer moet geaccepteerd worden en een juiste uitvoer genereren binnen een welbepaalde tijd. Er wordt enkel gekeken naar de uitvoer, de interne werking van het programma wordt niet beschouwd. Bij black box testen kijken we bijvoorbeeld of het programma aan de geëiste specificaties voldoet.

10.1.1 Testmethodes

Equivalence Partitioning
Het doel van equivalende partitioning is het aantal inputwaarden te reduceren tot het noodzakelijke minimum. Het is namelijk onmogelijk om alle inputwaarden te proberen. Het is nodig om de juist test cases te selecteren zodat alle mogelijke scenario's gecoverd worden.

Dit wordt hoofdzakelijk toegepast op inputs van een te testen component. We testen inputs met bereiken die geldig of ongeldig zijn. Bijvoorbeeld maand: $[1, 12]$ geldig, $]-\infty, 0]$ eerste ongeldige partitie $[13, +[$ tweede ongeldige partitie.

Er is slechts 1 test case nodig per partitie, de waarden binnen de partitie zijn equivalent (op gebied van behandeling door het programma). Als men denkt dat niet alle waarden binnen een partitie gelijk worden behandeld moet men deze opsplitsen in meerdere partities.

Opmerkingen

- Nooit meer dan 1 equivalentieklasse per test case
- Equivalence partitioning: geen stand alone methode om test case te bepalen
- Aanvullen met boundry value analysis: selecteren van meest effectieve waarden uit verschillende partities

Boundry-value Analysis
Het doel van *Boundry-value Analysis* is het bepalen van test cases die *off-by-one*-fouten dekken. Veel off-by-one fouten worden gemaakt bij de grenzen van inputbereiken. Andere veelvoorkomende fouten zijn, teller net te ver of niet ver genoeg laten komen en '<=' i.p.v. '<' of omgekeerd. We zoeken de grenzen via equivalence partitioning. Voorbeeld maanden: 0 en 1, 12 en 13. Er wordt steeds een clean en een dirty test case gemaakt. Bij de clean test case zorgt men voor een geldig resultaat van het programma, bij de dirty test case wil men net een correcte en gespecificeerde foutafhandeling verkrijgen. Er zijn 6 mogelijke testcase boundry-value Analysis:

- m, m-1, m+1 voor ondergrens
- n, n-1, n+1 voor bovengrens

Richtlijnen:

- Als een input bestaat uit een reeks getallen voer je de uiteinden in en getallen in die net buiten de reeks vallen.
- Als de input een aantal specificeert vul dan de minimum en maximum waarde in voor het aantal alsook net boven en onder het minimum en maximum.
- Hetzelfde geldt voor de outputs

Opmerkingen:

- Als er onmiddelijk een volgende bewerking uitgevoerd wordt, $k = i * j \rightarrow$ dan hoeven we enkel te kijken naar mogelijke waarden voor k en die waarden van i en j die die grenzen bepalen.
- Fouten die niet in de buurt van de grenzen liggen, zullen dus niet worden gedetecteerd.
- Soms niet mogelijk om voor bepaalde waarde de equivalence partitie(s) te bepalen.

10.1.2 Nadeel

Een groot nadeel is dat het onmogelijk is om alle mogelijke inputwaarden te testen (= uitputtend, exhaustief testen)

10.2 White box testen

Bij white box testen kijken we naar de structuur van het programma. Deze structuur wordt gebruikt bij het opstellen van de testcases. Een voorbeeld hiervan is *code inspection*[2]. Dit wordt vooral gebruikt voor *unit-testing*, maar is ook mogelijk voor integratie- en systeemtesten. Bij white box testen is het de bedoeling dat alle mogelijke paden worden doorlopen, alle logische beslissingen worden bekeken, alle lussen worden uitgevoerd op en binnen de grenzen en ook dat de datastructuren worden bekeken.

10.2.1 Waarom niet enkel black-box testen?

- Logische fouten en verkeerde veronderstellingen zijn omgekeerd evenredig met de kans dat een pad wordt uitgevoerd. De 'duistere' paden kunnen dus de meeste fouten bevatten, deze worden echter niet snel gevolgd bij *black-box testing*.
- Een pad kan regelmatig gevolgd worden, terwijl gedacht wordt dat dit weinig gebeurd.
- Typfouten geven evenveel kans op een obscuur als een hoofdpad.

10.2.2 Nadeel

Enkel de eigenschappen van de code worden gecontroleerd, maar niet de eigenschappen die de code zou moeten hebben. Er is niet te controleren of de code voldoet aan de specificaties en denkfouten worden niet gevonden.

10.3 Integratie testen

De integratietest is een softwaretest waarbij individuele softwaremodules verbonden worden en als een geheel getest worden. Deze fase komt na de ontwikkeltest met de unittest en voor de systeemtest. Voor de integratietest worden modules gebruikt die door de unittest gekomen zijn. Deze goedgekeurde modules worden aan elkaar gekoppeld en hier worden tests op losgelaten die in het integratie-testplan zijn beschreven. Als op deze manier alle submodules in een geheel zijn getest kunnen ze vervolgens door naar de systeemtest, of de acceptatietest waar weer andere testvormen worden toegepast.

Het doel van de integratietest is zekerheid te krijgen over de functionaliteit, performantie en betrouwbaarheid die van de submodulen gevraagd wordt. Deze submodules worden getest via hun interfaces waarbij gebruikgemaakt wordt van een blackboxtest. Testscripts worden afgespeeld met allerlei invoer en gekeken wordt of het geheel volgens verwachting reageert. Testcases worden gemaakt waarmee getest wordt of alle componenten binnen het geheel goed samenwerken, bijvoorbeeld bij een remote procedure call. Het geheel wordt opgebouwd uit goedgekeurde bouwstenen.

Om de submodules te testen moeten we beschikken over stubs en drivers. Drivers zijn modules die het gedrag van de module erboven te simuleren, stubs zijn dan weer modules die het gedrag van de module eronder simuleert. Dit is nodig om een (sub)module te apart te kunnen testen.

10.3.1 Bottom up

Bij bottom up gaan we eerst de onderste modules testen. Dit wordt best toegepast wanneer de belangrijkste of meest complexe modules onderaan liggen.
Voordelen:

- Het schrijven van drivers is meestal gemakkelijker dan het schrijven van stubs
- De onderste modules worden meest getest
- De drivers staan steeds rechtstreeks boven de te testen module → gemakkelijker om juiste testdata te hebben

Nadeel:
Er is geen modelprogramma tot dat alle onderdelen getest zijn.

[2] *Code inspection* is het doorlopen van de code op zoek naar fouten

10.3.2 Top down

Bij top down worden eerst de bovenste modules getest (deze zijn meestal UI's). Deze aanpak wordt best gebruikt wanneer complexiteit van het systeem bovenaan ligt.
Voordeel:

- Mogelijk om snel een werkende versie van het programma aan de eindgebruiker te tonen (sommige functies via stubs)

Nadelen:

- Soms onmogelijk om sommige tests van onderliggende modules uit te voeren (niet gebruikt door bovenliggende)
- Soms vrij moeilijk om aan bovenliggende module juiste testdata te geven zodat onderliggende goed getest wordt
- Afstand tussen module met invoer en module met uitvoer kan groot zijn → moeilijk om uit te maken wat er zich tussendoor afspeelt
- Alle modules grondig testen, vaak veel werk om stubs te maken → bepaalde testcases worden vlug sneller afgehandeld of overgeslagen

10.3.3 Big Bang

Bij de *Big Bang* methode wordt elke module apart getest om daarna alles samen voegen.
Nadeel:
Er komen een groot aantal fouten voor in elke mogelijke interactie tussen verschillende modules, hier is het moeilijk te bepalen welke module(s) de fouten veroorzaken.

10.3.4 Sandwich

Dit is een combinatie van top down en bottom up. De integratie begint zowel onderaan als bovenaan. De voordelen van beide worden hier gecombineerd.

11 Wat is all pairs testing? Illustreer met een voorbeeld.

Dit is een combinatorische software testmethode. Voor elke paar inputs worden alle mogelijke discrete combinaties bekeken. Door parallellisatie verkrijgen we typisch O($n * m$), m, n: grootst aantal mogelijkheden voor alle parameters redenering:

- De simpelste fouten worden veroorzaakt door 1 input parameter
- Volgende door interactie tussen paren parameters
- Fouten veroorzaakt door drie of meer parameters die interageren, komen minder voor. Het zou zeer duur zijn om deze via exhaustief zoeken te vinden

Voorbeeld:
We testen in parallel via een combinatietabel. Stel we hebben drie variabelen V1, V2 en V3 met mogelijke waarden (A,B,C), (X,Y) en (0,1).

We maken een tabel met voor elke variabele een kolom. De variabele met de meeste mogelijke waarden, komt terecht in de eerste kolom. Deze bepaalt tevens ook het aantal rijen. Elke mogelijke waarde van deze variabele zetten we `aantalMogelijkeWaardenTweedeGrootsteVar` keer uit als rij, telkens met een lege rij tussen.
Hierna vullen we alle mogelijke waarden in zoals weergegeven in de tabellen hieronder. Als we het voorbeeld uitbreiden met V4 en V5 met hun mogelijke waarden (E,F) en (G,H) zien we dat er nog altijd alle combinaties gemaakt kunnen worden. Als we echter V6 toevoegen (I,J) zien we dat we extra rijen nodig hebben, hiervoor dienen de blanke rijen.

V1	V2
A	X
A	Y
B	X
B	Y
C	X
C	Y

V1	V2	V3
A	X	1
A	Y	0
B	X	0
B	Y	1
C	X	1
C	Y	0

V1	V2	V3	V4	V5
A	X	1	E	G
A	Y	0	F	H
B	X	0	F	H
B	Y	1	E	G
C	X	1	F	H
C	Y	0	E	G

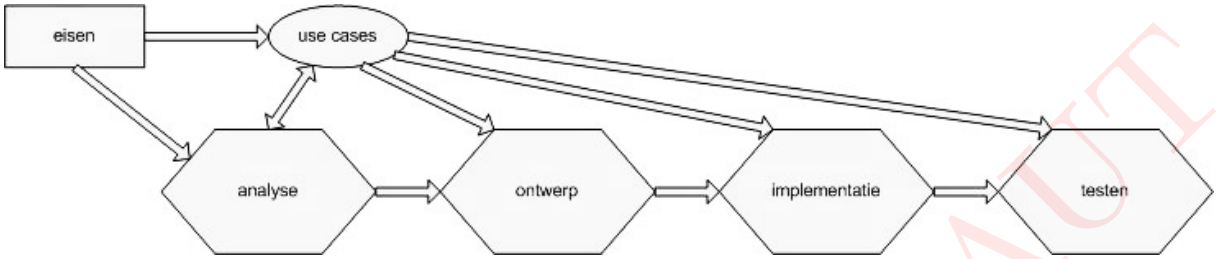
V1	V2	V3	V4	V5	V6
A	X	1	E	G	I
A	Y	0	F	H	J
		0			J
B	X	0	F	H	I
B	Y	1	E	G	J
				F	I
C	X	1	F	H	J
C	Y	0	E	G	I
				E	F

12 Bespreek het Unified Proces en Extreme Programming.

12.1 Unified proces [RUP]

De *Unified Software Development Process* of *Unified Process* is een populaire iteratieve en incrementele software development process framework. Deze is overigens use-case driven met een centrische architectuur. De levenscyclus van het proces bestaat uit een aantal cycli, elke cyclus produceert een nieuwe release (van het product) naar de klant, onder de vorm van een *executable*.

12.1.1 use case driven



De ontwikkelaars moeten nagaan of alle opeenvolgende modellen conform de use cases zijn. Bij de implementatie moet men steeds de gevraagde functionaliteit in het achterhoofd houden. Testers moeten de implementatie testen om zeker te zijn dat de use case correct geïmplementeerd zijn en dat het systeem dus voldoet aan de eisen van de klant.

12.1.2 centrische architectuur

De architectuur is de opbouw van systeem waarbij men zich moet afvragen wat de verschillende onderdelen, relaties en interacties tussen de onderdelen zijn.

Het is belangrijk het systeem logisch te verdelen in subsystemen waarbij de afhankelijkheden tussen subsystemen eenvoudig en beperkt zijn. De architectuur groeit uit de noden die de use cases bevatten. De architectuur wordt beïnvloed door het platform waarop het moet draaien (OS, DBMS, netwerkprotocollen, ...), de aanwezige herbruikbare delen (GUI, communicatie) en de niet-functionele eisen (performantie, betrouwbaarheid).

Hierbij hebben we te maken met het probleem van het kip en de ei, vermits de architectuur en use cases in parallel met elkaar staan. Men begint met een ruwe architectuur gebaseerd op algemeen verstaan van de use cases. Een subset van use case worden in meer detail gerealiseerd in termen van subsystemen, klassen en componenten. De systeemarchitectuur en de use cases worden volwassener met verloop levenscyclus, met een aantal verfijningen tot het stabiel wordt.

12.1.3 iteratief en incrementeel

Elk mini-project bestaat uit een iteratie[3] die resulteert in increment[4].

Iteratie
Bij een iteratie pakken we een groep van use cases aan die het product zal uitbreiden. Het behandelt de meeste risico's want vroegere iteraties moeten de grote problemen aanpakken, deze mogen niet worden uitgesteld! Vroege iteraties mogen ook prototypes zijn die eventueel worden weggegooid. Het mini-project zal vanaf de use case als volgt worden doorlopen: analyse, ontwerp, implementatie en dan vervolgens test.

Increment
Bij het increment wordt er extra functionaliteit toegevoegd, een model en document opgemaakt. Het moet niet noodzakelijk additief zijn, vervanging is ook mogelijk (eerst eenvoudig algoritme, dan meer gesofisticeerd).

[3] Een iteratie is een aantal stappen die in de workflow worden geselecteerd en worden uitgevoerd.
[4] Increment is de groei van het product

12.2 Extreme programming [XP]

12.2.1 kenmerken

- alle projectmedewerkers verstaan ten gronde wat de gebruiker(s) wil(len)
- ontwikkel software incrementeel
- herbekijk de eisen na elke kleine stap
- lever kleine incrementele componenten elke 3 weken
- bewijs dat zonder defecten (nieuwe term voor bug)
- herbekijk alle vragen van klant elke 3 weken
- werkt enkel goed voor kleine projecten

12.2.2 Common Sense to the Extreme

12.2.3 Vereisten

- iedereen ontwerpt elke dag
- altijd het eenvoudigste kiezen
- code steeds herzien
- altijd testen
- ontwerp en verfijn steeds de architectuur
- nieuwe code elke dag integreren en testen
- elke dag afleveren
- klant/gebruiker moet steeds beschikbaar zijn
  - voor verdere vragen van ontwikkelaars
  - voor evaluatie, elke dag
- *pair programming*
- code van iedereen → iedereen mag wijzigen
- elke dag half uurtje samen werk te verdelen
- optimale teamgrootte: 6 (max 12)
- optimale projectlengte: 6-9 maanden

12.2.4 nadelen

- planning max 3 weken vooruit → onmogelijk om gehele kostprijs van project op voorhand te schatten (hoe projecten plannen?)
- werken aan kleine blokjes en onmiddellijke integratie in groot geheel → structuur van groot geheel?
- gebruikers/klant steeds beschikbaar

13 Hoe ga je kwaliteit van software beoordelen? Wat zeggen CMM en ISO 9000 hierover?

13.1 Software kwaliteit

Software kwalitiet is conform aan het expliciet vermelde functionele en perfomantie-vereisten, expliciet gedocumenteerde ontwikkelingsstandaarden, en impliciete karakteristieken die van alle professioneel ontwikkelde software wordt verwacht.

3 belangrijke punten

- voldoen aan expliciete vereisten (bv gevraagde printfaciliteit ontbreekt of werkt niet naar behoren → kwaliteit daalt)
- niet volgen van ontwikkelingscriteria (vastgelegd in standaard) → meestal kwaliteitsverlies (bv controle model voldoet aan functionele vereisten)
- voldoen aan impliciete eisen zoals bv onderhoudbaarheid

13.1.1 Beoordelen kwaliteit

Kwaliteit kan beoordeeld worden op volgende zaken:

- correctheid (correctness): formele 1 tot 1 correspondentie tussen software product en zijn functionele specificatie. Alles moet wiskundig neergeschreven worden.
  - ? niet mogelijk om correctheid te bewijzen in zulke complexe producten als software
  - ? functionele specificaties zelden precies en stabiel genoeg
- betrouwbaarheid (reliability of dependability): software gedraagt zich goed en zoals gebruiker verwacht
  - in software engineering: betrouwbare software mag "gekende"fouten fouten bevatten (zelfs met nieuwe fouten soms nog betrouwbaar genoeg)
  - telkens zelfde gevolg van eenzelfde actie, telkens crashen wanneer er op een knopje duwt → zeer betrouwbaar
- robustheid (robustness): onwaarschijnlijk dat software faalt (crasht) of onherstelbaar faalt
- correctheid, betrouwbaarheid en robuustheid
  - voldoet software aan gewenste functies, zoals verwacht
  - betrekking op product en proceskwaliteit
- performantie (performance):
  - binnen vooropgestelde doelen, meestal response time;
  - zwart-wit-kwaliteit
- bruikbaarheid (usability):
  - user friendliness: hoe gemakkelijk is het voor gebruikers om software voor hen te laten werken
  - zeer subjectief
  - vooral voor user interface ontwerp
  - hoe meer standaard hoe meer bruikbaar
- verstaanbaarheid (understandability)
  - interne structuur en gedrag van software
  - voorwaarde voor onderhoudbaarheid
- onderhoudbaarheid (maintainability, repairability=1)
  - corrigeren van fouten en onvolkomenheden
  - aanpassen aan nieuwe eisen

  - software verbeteren om nieuwe kwaliteiten te geven
- schaalbaarheid (scalability of evolvability) gemak waarmee software groeiende vraag kan volgen
  - enkel als duidelijke en verstaanbare architectuur
  - Afwijken verstaanbaarheid, enkel voor performantie
  - Vaak werken met abstracte klassen → nieuwe implementaties toe te voegen

verstaanbaarheid + onderhoudbaarheid + schaalbaarheid = supportability

- herbruikbaarheid (reusability): in hoever kunnen componenten herbruikt worden
  - als component in ander product
  - als generisch framework dat nog aan nieuwe product moet aangepast worden
  - slaat op product en proces
- overdraagbaarheid (portability): verschillende hardware/software-platformen, zonder aanpassingen of met kleine aanpassingen/parameters
- (interoperability):
  - samenleven of samenwerken met andere software (die mogelijks nog niet bestaat)
  - open systemen
- productiviteit (productivity)
  - proceskwaliteit: efficiëntie en perfomantie van proces
  - snelheid waarmee software wordt geproduceerd (gegeven aantal resources). We kunnen de productiviteit verhogen door zoveel mogelijk code te laten genereren.

  Voorbeeld: GUI tekenen → code genereren voor verschillende platformen.
- (timeliness)
  - proceskwaliteit mogelijkheid om software op tijd te leveren
    * volgens oorspronkelijk plan
    * on-time-to-market (commerciële software)
- zichtbaarheid (visibility)
  - proceskwaliteit
  - transparant proces: duidelijk gedefinieerde en gedocumenteerde stappen en activiteiten
  - vereiste voor CMM en ISO

13.1.2 Meetbare kwaliteit

- Auditability: hoe gemakkelijk kan gechecked worden of conform standaards
- Accuracy: precisie van berekeningen en controle
- Communication commonality: mate waarin standaard interfaces, protocollen bandbreedtes wordt gebruikt
- Conciseness: compactheid in aantal lijnen code
- Consistency: gebruik van uniforme ontwikkelings- and documentatie-technieken
- Data commonality: gebruik van standaard datastructuren en types
- Error tolerance: schade als programma fout tegenkomt
- Execution efficiency: run-time performantie van programma
- Generality: breedte van mogelijke toepassingen van programmacomponenten

- Hardware independence: graad waarin software ontkoppeld is van hardware waarop het loopt
- Instrumentation: graad waarin programma zichzelf in de gaten houdt en fouten identificeert
- Modularity: functionele onafhankelijkheid van programmacomponenten
- Operability: gemak van gebruik van programma
- Security: beschikbaarheid van mechanismen om programma's en data te controleren en te beveiligen
- Self-documentation: graad waarin source code betekenisvolle documentatie levert
- Simplicity: hoe gemakkelijk een programma verstaan kan worden
- Software system independence: hoe onafhankelijk is programma van
  - niet-standaard kenmerken van programmeertaal
  - karakteristieken van besturingssysteem
  - andere omgevingsbeperkingen
- Traceability: mogelijkheid om een ontwerp of programmacomponent terug te brengen tot requirements Training: hoe goed helpt de software nieuwe gebruikers met het systeem omgaan.

13.2 ISO 9000

Het ISO 9000 label wordt gebruikt voor alle industriën, niet enkel software. ISO 9001 is het meest algemeen: ontwerp, ontwikkeling en onderhoud van producten. ISO 9000-3 interpreteert ISO 9001 voor software ontwikkeling. ISO 9000 legt geen kwaliteitsproces vast. Het is dus zo dat we niet noodzakelijk betere software produceren, want er is geen aandacht voor de "best practices"te reflecteren. Het is mogelijk om bv testprocedure te definiëren die leidt tot incomplete software testing, maar zolang het bedrijf procedures volgt en documenteert, is het conform de standaard.

interessante site:

- `http://www.praxiom.com/`
- `http://www.isoeasy.org/`
- `http://www.isoeasy.org/jokes.htm`
- ISO: International Standardization Organisation (Zwitserland)

13.2.1 versie 2000

- Focus op klanten
  - versta hun noden
  - voldoe aan hun eisen
  - maaovertref hun verwachtingen
- Leiderschap
  - geef eenduidige richting aan
  - creëer omgeving die mensen aanmoedigt om
  - objectieven organisatie te bereiken
- Betrek uw mensen erbij
  - op alle niveau's
  - zorg dat mensen hun mogelijkheden gebruiken en zich verder ontwikkelen
- Gebruik processen
- Neem een systeembenadering (behandel gerelateerde processen als 1 systeem)
- Moedig continue verbeteringen aan
- Verzamel feiten voor nemen van beslissingen
- Werk met uw leveranciers, onderhoud mutueel-profijt-relatie

13.3 CMM: Capability Maturity Model

13.3.1 Rol van maturiteitsmodel

- levert mogelijkheid om de softwareprocescapaciteit van een onderneming te meten
- zet doelen en prioriteiten voor procesverbetering
- helpt om acties te plannen
- geeft een methode om procesmanagement en kwaliteitsverbeteringsconcepten toe te passen op softwareontwikkeling en onderhoud
- leidt een organisatie naar software engineering excellence

13.3.2 karakteristieken van immature proces

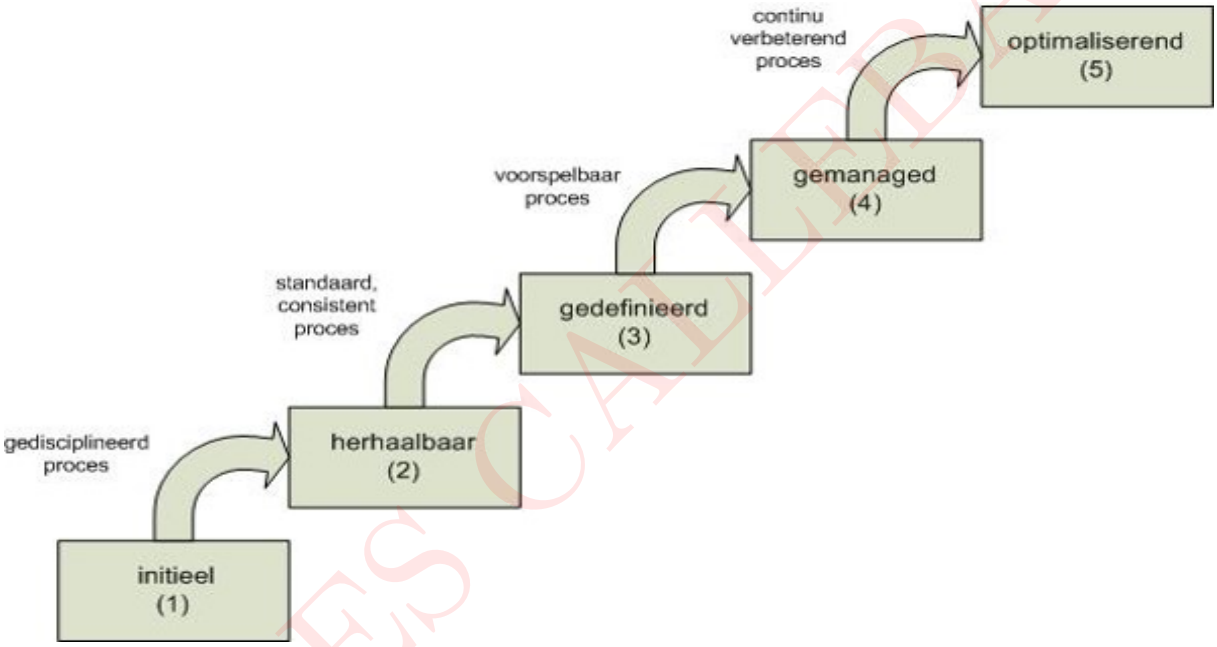
- ad hoc: geïmproviseerd proces door ontwikkelaars en hun management
- niet nauwgezet gevolgd en afgedwongen
- zwaar afhankelijk van huidige ontwikkelaars
- moeilijk om productkwaliteit te voorspellen
- door minder goede schattingen, grote kans op kost en planningsproblemen
- vaak productfunctionaliteit en kwaliteit afgebrokkeld ten voordele van halen planning
- gebruiken van nieuwe technologie zeer risicovol

13.3.3 Karakteristieken van mature proces

- gedefinieerd en gedocumenteerd
  - verstaan
  - gebruikt
  - levend
- zichtbaar ondersteund door management
- rollen en verantwoordelijkheden goed gedefinieerd en verstaan
- trouwheid aan proces wordt nagekeken en afgedwongen
- consistent met manier van werken
- gemeten
- ondersteund door technology

13.4 Vergelijking ISO 9000 en CMM

Als het bedrijf ISO 9001 compliant is dan voldoet het aan CMM-niveau 2. Het omgekeerde (CMM-niveau 2 naar ISO 9001) blijkt niet zo makkelijk te zijn. De algemene principes zijn analoog, sommige zaken worden enkel in 9001 bekeken en anderen enkel in CMM. Het is gemakkelijker om vanuit CMM een ISO 9001 te halen dan omgekeerd.



14 Bespreek grondig het Singleton patroon.

Singleton is een creatie ontwerppatroon om het aantal objecten van een bepaalde klasse tot één te beperken. Met dit ontwerppatroon is het mogelijk om de toegang tot bepaalde systeembronnen altijd via één object te laten gaan.

Een toepassing van de singleton is bijvoorbeeld het maken van unieke identificatienummers binnen een programma. Om er altijd zeker van te zijn dat elk identificatienummer uniek is, is het handig om dit door één enkel object te laten genereren. Dit is dan een singleton.

Een singleton wordt gemaakt door een klasse te definiëren met een methode die een nieuw object aanmaakt als het nog niet bestaat en een bestaand object teruggeeft als er al wel een dergelijk object bestaat.

Een andere toepassing is logging:

```
public class Logger{
    private static Logger logger = null;
    private PrintWriter pw;

    private Logger(){
        try {
            pw = new PrintWriter(new FileWriter("log.txt"));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public void finalize(){
        try {
            pw.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public synchronized void Log(String tekst){
        pw.println(tekst);
    }

/*
    syncrhonized is nodig vermits 2 verschillende threads
    in de if ... null kunnen zitten en als hierachter de ene thread (1) stopt
    en de andere (2) gaat verder dan lijkt het of er nog geen instantie
    is aangemaakt dus krijgt thread 2 een object.
    Hierna gaat Thread 1 weer verder en krijgt ook een instantie
*/
    public static synchronized Logger getLogger(){
        if(logger==null) logger = new Logger();
        return logger;
    }
}
```

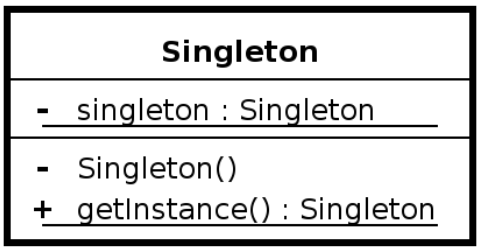
14.1 Problemen

We krijgen te maken met performantie-problemen. Een methode synchronized maken kan de performantie sterk reduceren. Als de performantie niet kritisch is dan gebruik je best synchronized. In het andere geval is het beste dat je onmiddellijk een instantie maakt i.p.v. het uit te stellen tot als het nodig is. Het nadeel van deze methode is dat het opstarten van het programma iets langer kan duren en indien er geen instantie wordt gevraagd, dat hij toch al is gemaakt.

```
public class Logger{
    private static Logger logger = new Logger();

        // de rest van de code

    public static Logger getLogger(){
        return logger;
    }
}
```



Een andere mogelijkheid om de performantie te verbeteren is enkel synchronized te zetten bij de eerste oproep:

```
public class Logger{
    private static Logger logger = null;

        // de rest van de code

    public static Logger getLogger(){
        if(logger==null){
                synchronized(Logger.class) logger = new Logger();
        }
        return logger;
    }
}
```

Java laat ook creatie van objecten via cloning toe. Als een singleton-class final is en rechtstreeks afgeleid is van Object dan blijft `clone() protected` en is er geen verdere overerving mogelijk. Als echter de singleton-class erft van een klasse die clone() als public heeft overschreven en die `Cloneable` implementeert moet je de `clone()` overschrijven en `CloneNotSupportedException` opgooien. Een slechte manier om dit op te lossen is this returnen, maar dat is de klant bedriegen en is niet waterdicht als je reflectie gebruikt.

15 Bespreek de Strategy, Observer en Chain of Responsibility patronen.

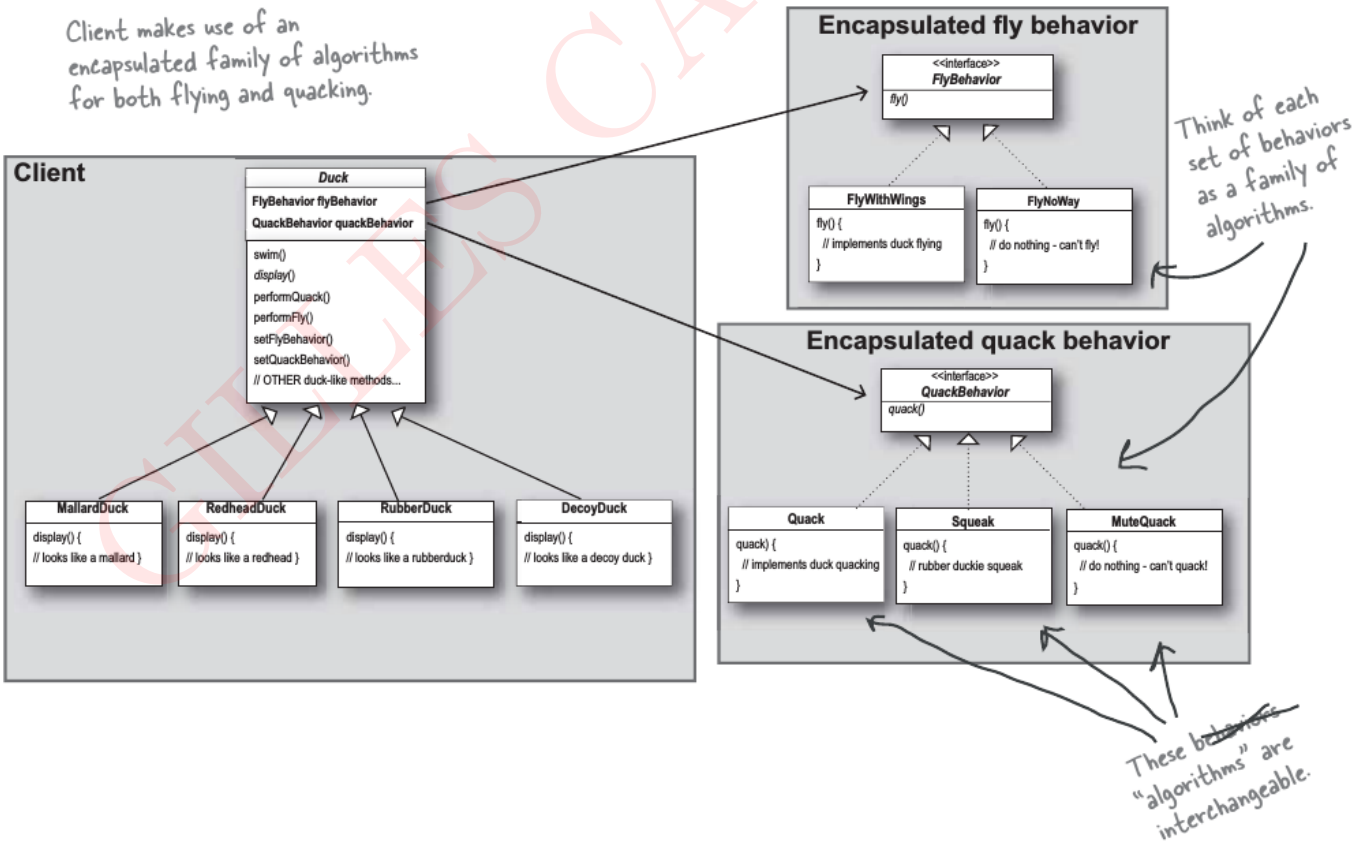
Deze patronen zijn gedragspatronen.

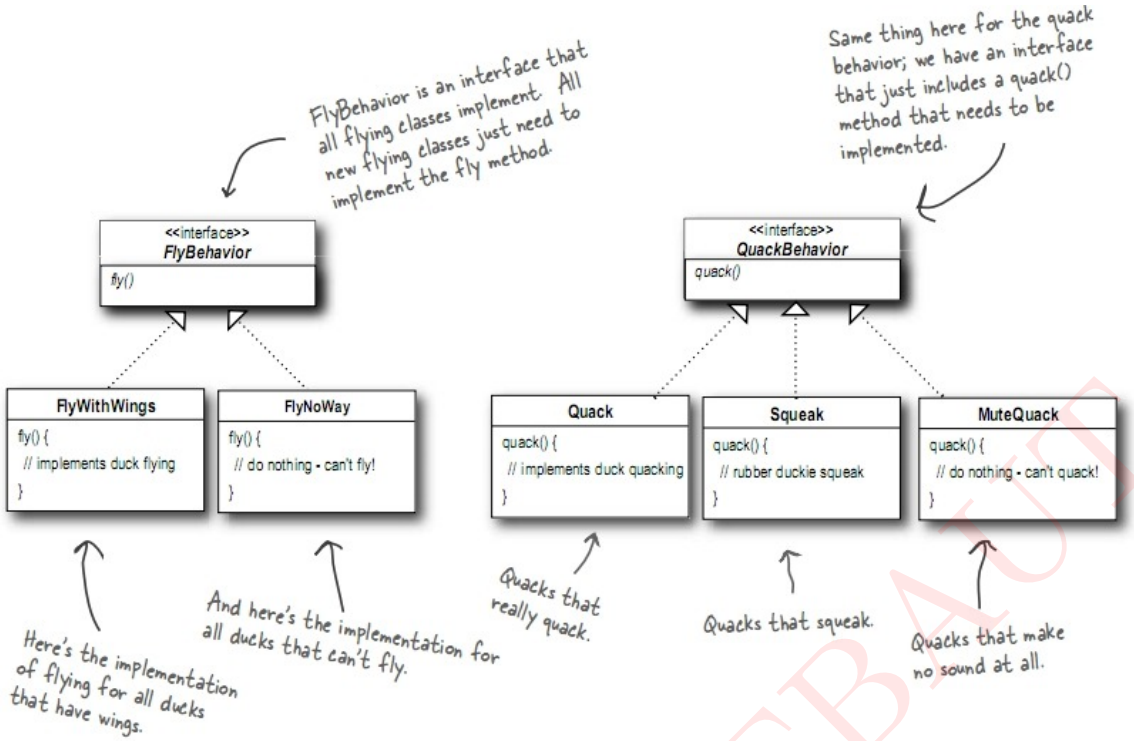
15.1 Strategy Patroon

Door het strategy patroon kunnen we gedragingen dynamisch (at runtime) veranderen of toewijzen. Als we gedragingen willen aanpassen moeten we hierdoor niet heel de klasse aanpassen. De gedragingen die kunnen veranderen staan in compositie met interfaces, waardoor het hergebruiken van gedragingen nu mogelijk is.

15.1.1 Voorstellen

- Polymorfisme
  De hoofdklasse `Duck abstract` maken met gemeenschappelijke operaties en overschrijven wat er veranderd. Nadeel: we zien dat er in de hoofdklasse niet enkel meer gemeenschappelijke operaties zijn. Een rubberen eend kan namelijk niet kwaken, maar wel piepen.
- Overschrijven
  Een oplossing voor het vorig probleem is de operaties die niet worden gebruikt of niet van toepassing zijn overschrijven.
  We zien dat we een probleem hebben met de (statische) overerving vermits niet alles in de `Duck-klasse` moet overgeërfd worden.
- Oplossing
  We moeten de gedragingen die veranderen encapsuleren, scheiden van de rest van de code. We maken een klasse om het gedrag te encapsuleren. We scheiden het veranderlijke van wat hetzelfde blijft.



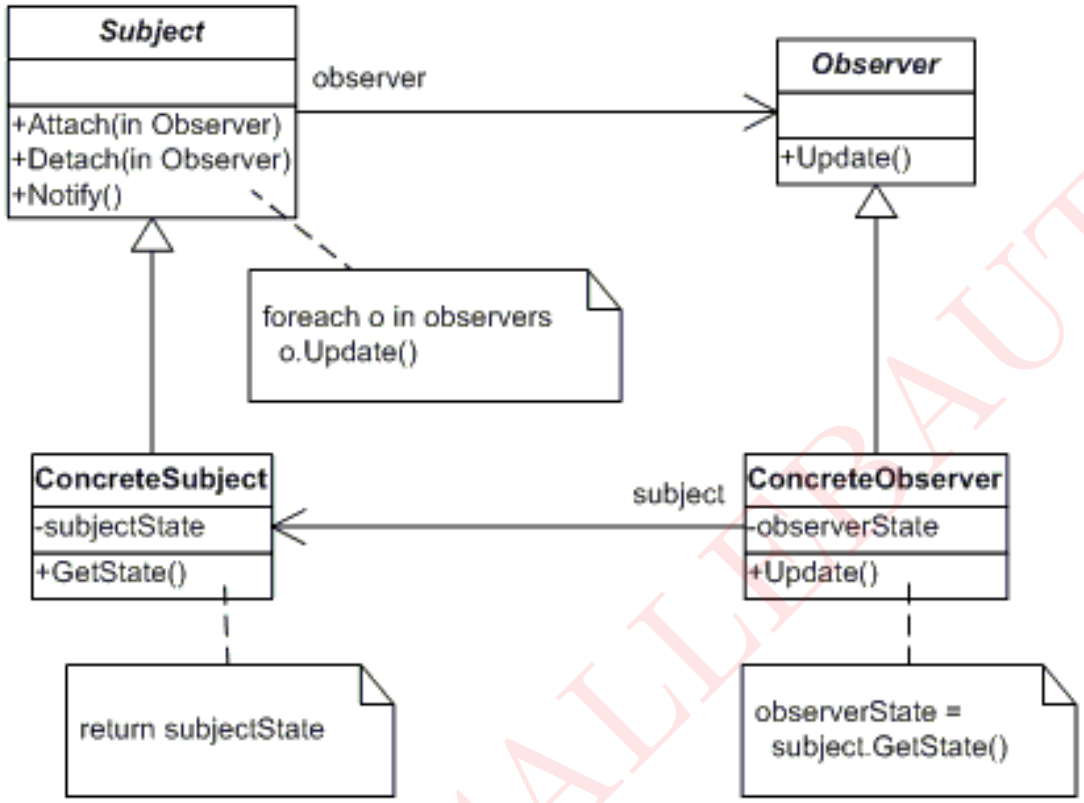


15.1.2 Voordelen en resultaat

- We kunnen gedragingen toevoegen zonder `Duck` te veranderen
- We kunnen gedragingen nu ook hergebruiken
- Gedragingen kunnen at runtime geselecteerd worden
- Compositie is flexibeler dan overerving
- Gedragingen worden afgeschermd en kunnen dynamisch worden aangepast

15.2 Observer Patroon

Het observer patroon definieert een één-op-veel afhankelijkheid tussen objecten, zodat als 1 object wijzigt, alle afhankelijke objecten gewaarschuwd en ge-update worden. Het patroon beschrijft een efficiënte en redelijk ontkoppelde manier waarop objecten in een programma kennis kunnen nemen van relevante toestandsveranderingen binnen andere objecten in hetzelfde programma.



Subject en Observer zijn beide interfaces, losse koppeling.

15.2.1 Observer

De referenties naar de subjecten worden meestal in de main bijgehouden. Als er een update is dan wordt het subject meegegeven als paramater aan de observer.

15.2.2 Subject

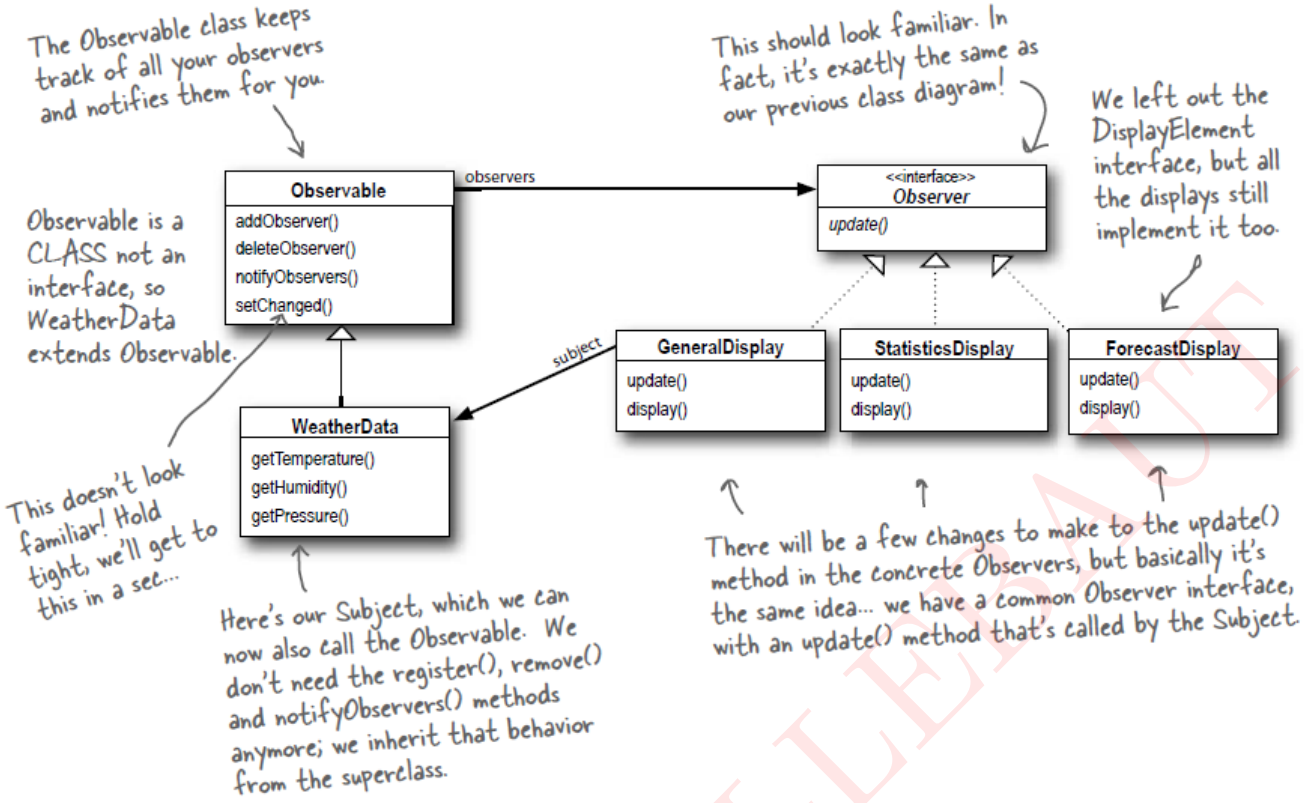
Subject heeft een lijst van Observers, een Observer wordt toegevoegd bij het registreren. Het Subject object kan de Observers verwittigen, maar wanneer?
Als het subject moet verwittigen dan roept het Subject de `update()` methodes aan van elke observer. Hierbij kunnen twee modellen worden gehanteerd:

- pull-model
  De observer trekt data uit het subject, en moet zelf uitvissen wat er veranderd is. Om dit te kunnen moet hij de interface van het subject kennen. (`subject.getState()`)
- push-model
  De observer krijgt data van het subject als hij verwittigd wordt. De observer moet de data kunnen interpreteren (casting). Dit is minder herbruikbaar en is strakker gekoppeld.

15.2.3 Voordelen

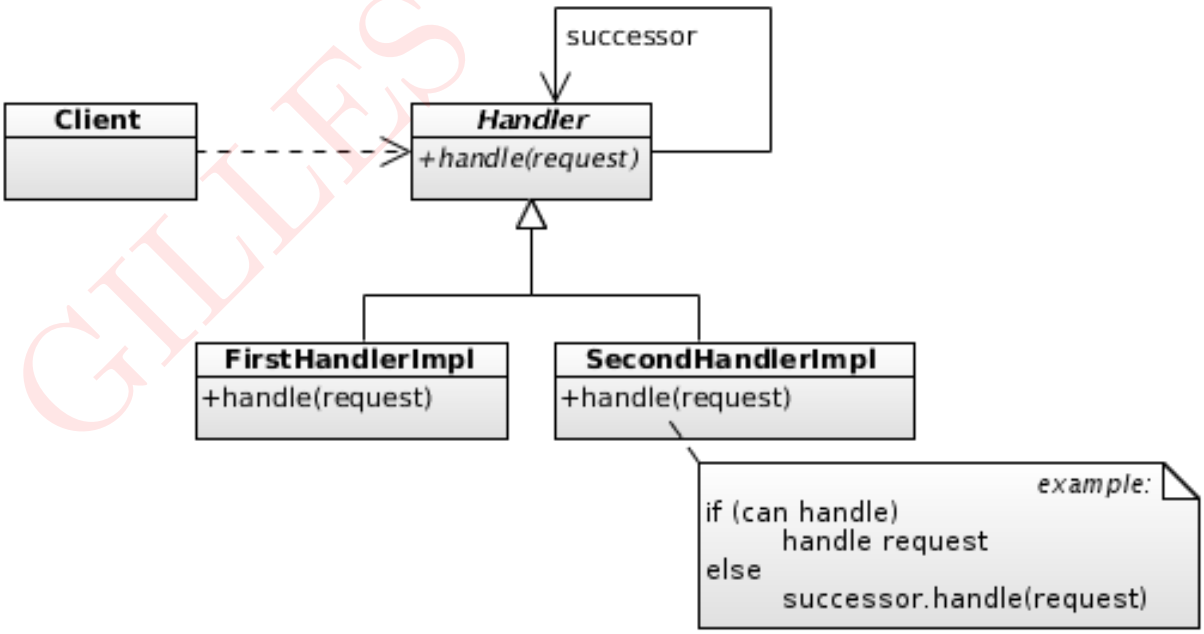
- implementatiedetails zijn afgeschermd aan beide zijden
- details kunnen variën in de concrete klassen
- er kunnen nieuwe Observers worden toegevoegd

- het Subject verandert niet als er nieuwe Observers bijkomen



15.3 Chain of Responsibility Patroon

Het CoR design patroon laat een object toe een commando te zenden zonder te weten wie het zal ontvangen en afhandelen.



- `Handler`
  Definieert een interface voor het behandelen van requests

- `RequestHandler`
  Behandelt de request waarvoor het verantwoordelijk is. Als hij de request niet kan verwerken zendt hij deze door naar zijn opvolger.

- `Client`
  De Client zend het commando naar het eerste object in the *chain* dat het commando zou kunnen afhandelen.

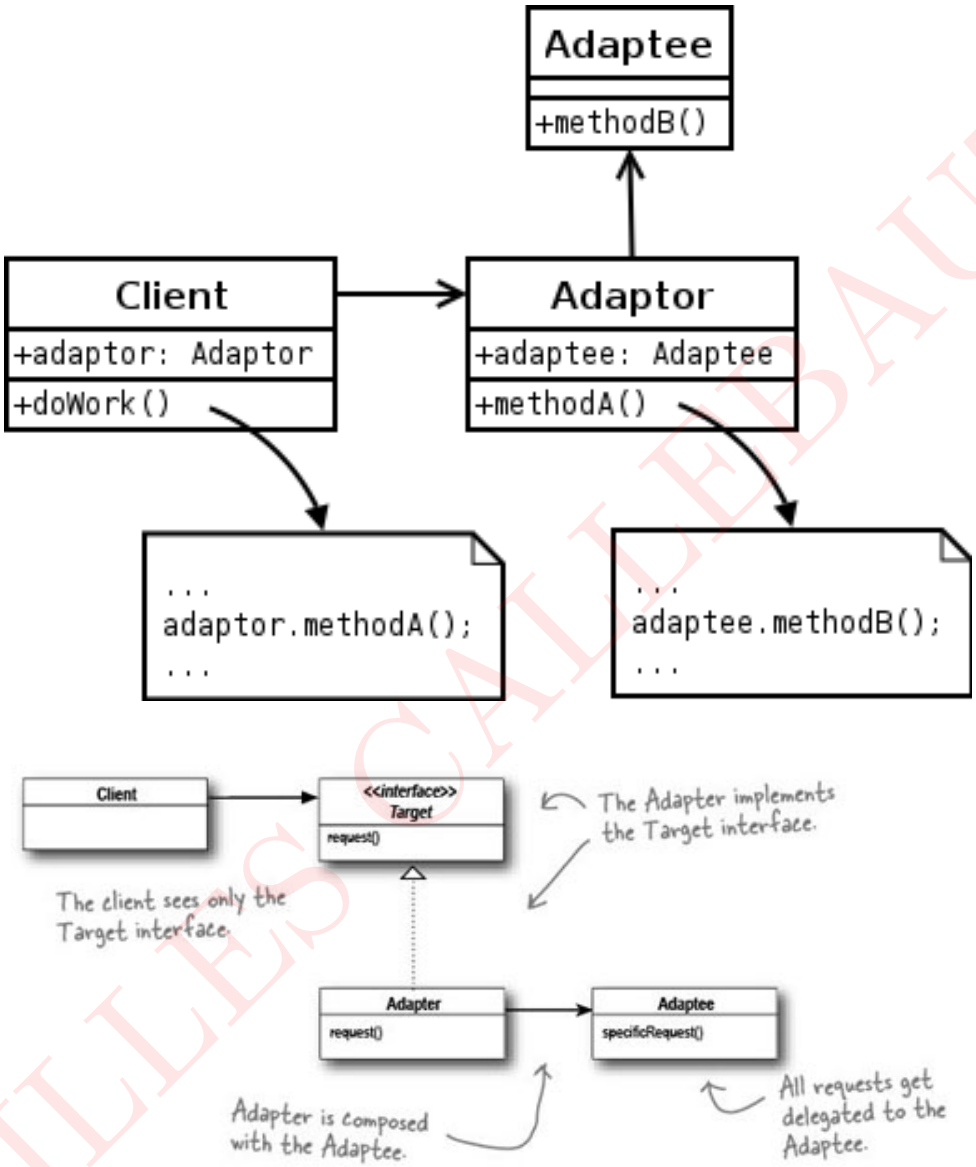
CoR wordt bijvoorbeeld gebruikt bij exception handling. We gebruiken CoR als je de request wilt ontkoppelen van zender en ontvanger. Als er verschillende objecten zijn die deze request kunnen afhandelen, waarbij het niet uitmaakt wie deze afhandelt.

16 Bespreek de Adapter, Facade en Decorator patronen.

Deze patronen behoren tot de structurele ontwerppatronen.

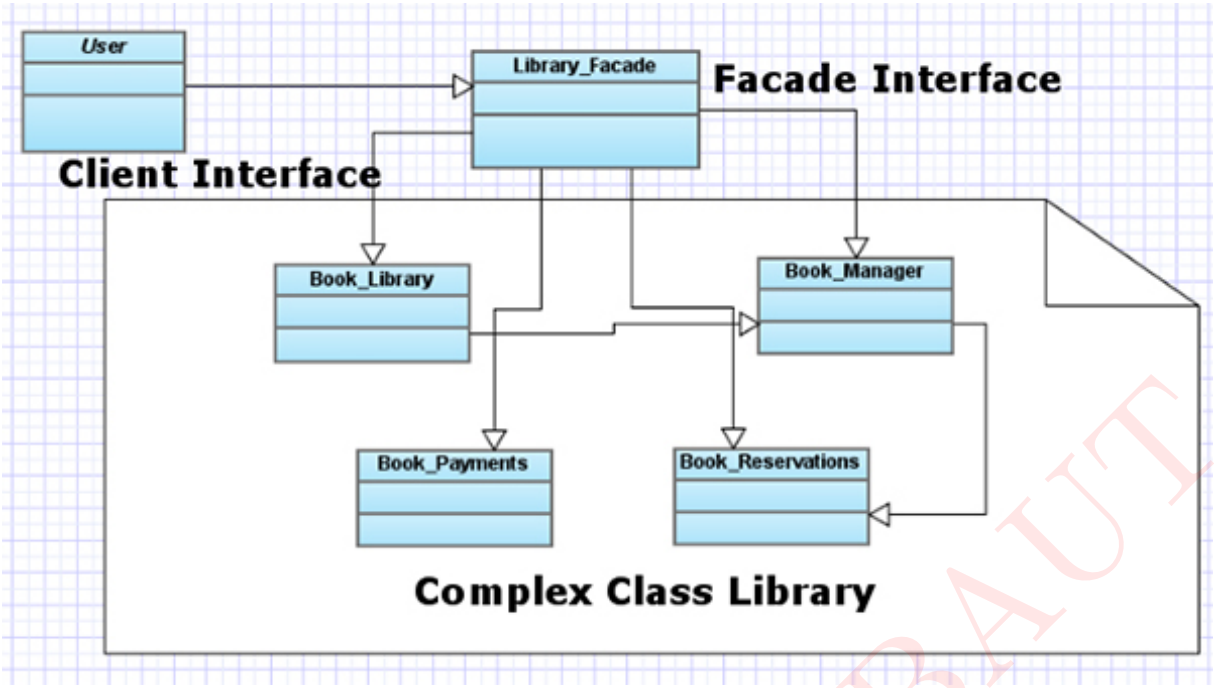
16.1 Adapter Patroon

De bedoeling van dit patroon is de interface van een klasse om te zetten naar een interface dat de *client* verwacht. Dit gebeurt door een klasse te maken die de verwachte interface implementeert maar de *client calls* omzet naar de nieuwe interface.



16.2 Facade Patroon

Een **adapter vertaalt**, een **facade** daarentegen gaat **vereenvoudigen**. Een facade zorgt ervoor dat een bibiliotheek eenvoudiger te gebruiken/verstaan is. Een facade levert handige methoden voor veelvoorkomende taken, waardoor de code in de *client* gemakkelijker te lezen is. Vermits de facade vereenvoudigt zal de *client-code* minder afhankelijk zijn van de interne werking van de bibliotheek, wat de flexibiliteit ten goede komt. Voorbeeld: een afstandsbediening voor TV, DVD, ... Een facade moet dus een higher-level interface definiëren waardoor de subsystemen gemakkelijker in gebruik zijn. Deze subsystemen moeten niet met elkaar gebonden zijn, maar als je ze tezamen wilt gebruiken dan kies je best voor een facade. Een nadeel van een facade is dat je vrijheid moet afgeven vermits je enkel kan doen wat de facade vrijgeeft.

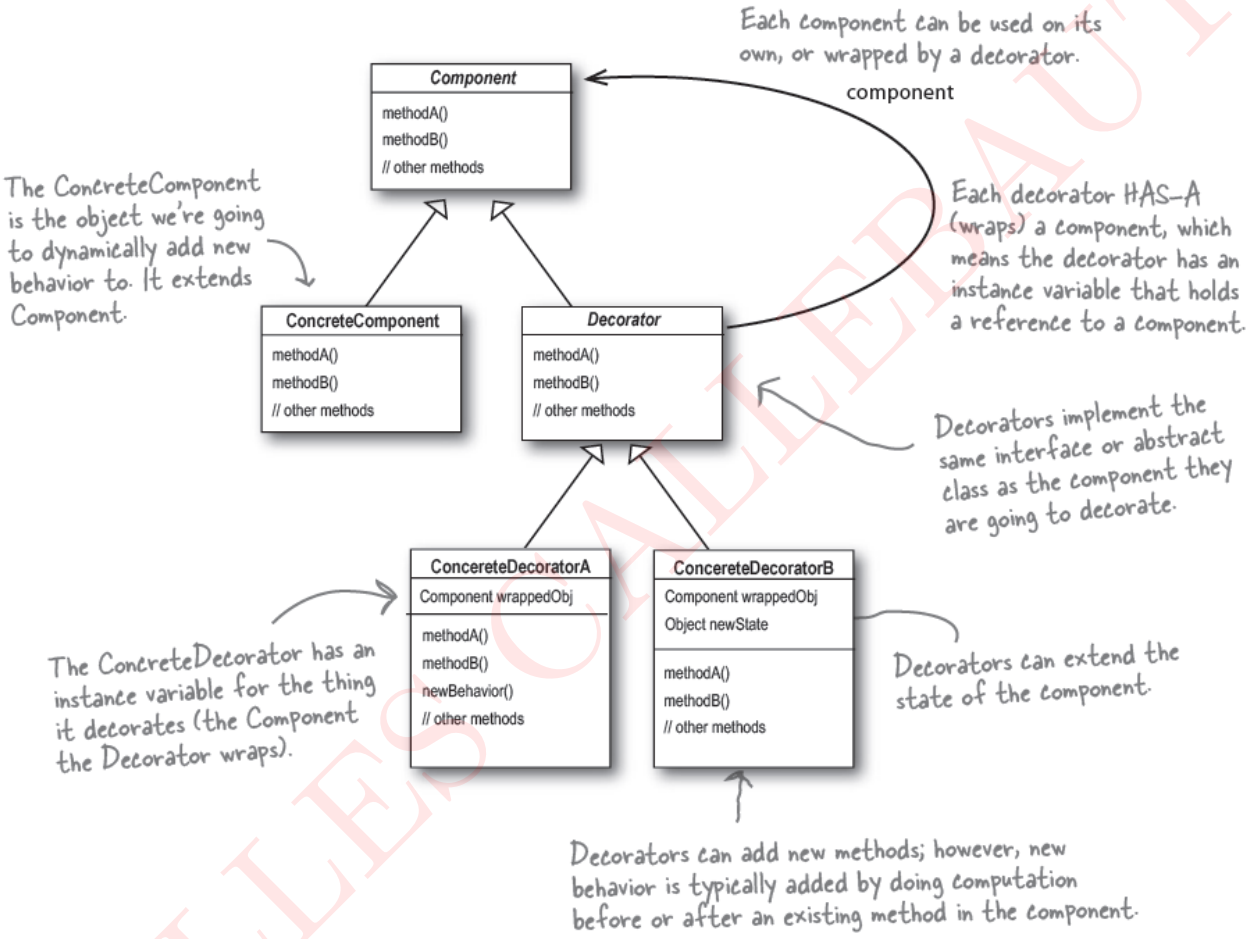


16.3 Decorator Patroon

Door het decorator patroon kunnen we dynamisch extra functionaliteit toevoegen aan een object. Dit is flexibeler dan uitbreiding van functionaliteit door middel van subklassen (overerving).

De Decorator implementeert dezelfde interface als de abstractie. Het beheert een concrete instantie van de abstractie via compositie, en voegt functionaliteiten toe. Het roept zijn objectmethoden op en combineert het resultaat met de toegevoegde functionaliteiten.

Voorbeelden van het decorator-patroon in Java is de I/O stream, `.readLine()` is toegevoegd door `BuffereInputStream` en `BufferedReader` en zit niet in de superklasse. Een ander voorbeeld is het opmaken van een GUI waarbij je verschillende componenten kunt toevoegen aan vensters, zoals *scrollbars*, andere venster,....



Figuur 12: Opmerking, compositie tussen Decorator en Component

17 Bespreek Aspect Oriented Programming

17.1 AOP

Aspectoriëntatie is bedoeld als antwoord op een probleem waar alle "klassieke"paradigma's mee kampen, namelijk de vraag hoe om te gaan met de zogeheten crosscutting concerns: handelingen die door het hele programma heen uitgevoerd moeten worden. Typische voorbeelden hiervan zijn logging en beveiliging: op vrijwel ieder punt in een gemiddeld programma bestaat de behoefte aan de mogelijkheid om zaken naar een log weg te schrijven om fouten te traceren en zeer veel programma's moeten op verschillende punten controleren of de gebruiker van het programma wel gerechtigd is om de opgevraagde handeling uit te voeren. De "klassieke"paradigma's hebben hiervoor geen makkelijke oplossing en vallen daarom terug op het meerdere malen opnemen van precies dezelfde code op iedere plaats waar dezelfde handeling uitgevoerd wordt. Met als resultaat dat een verandering aan de uitvoering van een dergelijk crosscutting concern betekent dat door het hele programma heen code moet worden aangepast.

Als oplossing hiervoor biedt AOD (Aspect Oriented Development) de mogelijkheid een stukje code te schrijven dat op een groot aantal, door de programmeur te definiëren punten, ingevoegd wordt. Dit invoegen gebeurt zonder dat de geschreven code waarin ingevoegd wordt, aangepast wordt op de invoeging (dat er op een gegeven plaats iets ingevoegd wordt, kan men dus niet zien aan de programmacode). Om dit voor elkaar te krijgen, geeft de programmeur extern aan de programmacode een aantal punten op waarin bepaalde code ingevoegd dient te worden (*joinpoints*). Een apart programma neemt de code van de programmeur en zijn aanwijzingen over invoegen en stelt daarmee een nieuw programma samen waarin de juiste code op de juiste plaats ingevoegd is.

Er zijn in principe twee tijdstippen waarop het weven (het samenvoegen van code) kan plaatsvinden: tijdens de compilatie van code naar programma (dit wordt statisch weven genoemd) en tijdens het uitvoeren van het gecompileerde programma (dynamisch weven). In het eerste geval leeft de AOD-uitbreiding op de bestaande techniek in de compiler, in het tweede geval wordt de uitbreiding extern aangebracht via een preprocessor. Het laatste bestaat uit 3 fasen: het compileren, laden en uitvoeren.

17.1.1 Termionologie

- Aspect
  Dit is een abstractie die een *concern* implementeert.
- Join point
  Een *joint point* is een plaats in het programma waar een aspect kan in verwoven worden.
- Pointcut
  Een *pointcut* definieert waar een aspect in het programma zal worden gewoven. Een *pointcut* bestaat uit een collectie van *join points*.

Er bestaan verschillende soorten *joint points*:

- method & constructor call
- method & constructor execution
- field get & set
- returning normally from a method call
- returning from a method call by throwing an error
- exception handler execution
- static & dynamic initialization

18 Beveiliging van software is enorm belangrijk. Bespreek een aantal problemen die bij het gebruik van webapplicaties kunnen voorkomen.

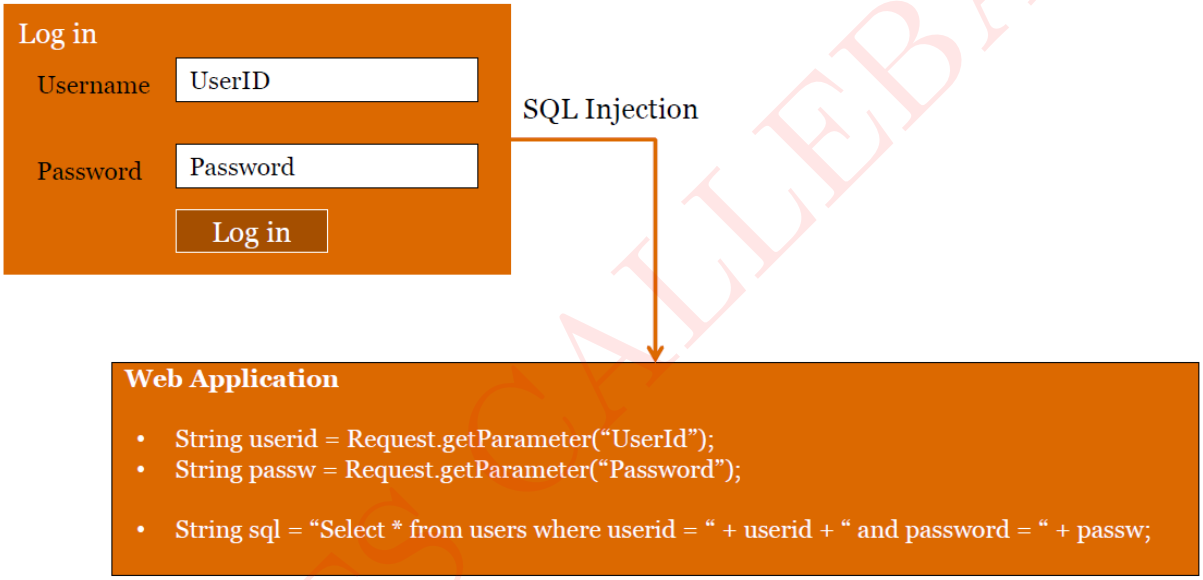
18.1 Misleidende assumpties

- Ik gebruik een Firewall dus ik ben veilig
  → attacks komen vaak binnen via poorten 80 of 443
- Ik gebruik SSL dus ik ben veilig
  → SSL beveiligd de connectie, niet de browser of de server.

18.2 Injection

Er kan misbruik worden gemaakt van interpretatie. Elk statement is een combinatie van een commando en data. Het idee achter injection is het omvormen van data tot commando's.

18.2.1 SQLi



Oplossingen:

- Parameterized input processing (PreparedStatments[5]
- Automatic Escaping
- Input validation

18.3 Broken authentication and sessions management

Dit wordt gebruikt om accounts te misbruiken, sessies te hijacken. Dit kan door fout gebruik van GET-methode, (paswoorden in URL, sessie-ID in URL, ...). Hierdoor kan de hacker zich voordoen als iemand anders.

Oplossingen:

- Gebruik één, geverifieerde authenticatie controle
- Betrouw op de *framework session management* (maar verifieer!)

[5] We maken eerst een template structuur van de query en zenden deze door naar de DB. `db->prepare("SELECT * FROM users where id=?");`, hierna zenden we de data apart van de statement, `db->prepare("SELECT * FROM users where id=?");`)

18.4 Cross site scripting

Rauwe data wordt teruggestuurd naar de client. Hierbij kunnen er verschillende types worden onderscheiden:

- Reflected (XSS direct terug in uitgevoerde respons)
- Stored (XSS wordt opgeslagen in bv. database en wordt telkens uitgevoerd wanneer hij wordt *ge-accessed*)
- (andere waren minder uitvoerig besproken)...

Wordt heel vaak via mail gedaan. De gebruiker moet dan op een link klikken (eerste oplossing: kijk naar de link!), in de payload die je terug van de server krijgt zit dan *malware* die dan in de browser van de gebruiker kan worden uitgevoerd. Nu heeft deze malware de volledige functionaliteit van JavaScript en kunnen er sessies worden gestolen, responsies worden aangepast, aan *phising* gedaan worden, ...

Oplossingen:

- Ouput encoding
  De bedoeling van output encoding is het omvormen van onbetrouwbare input naar een veilige vorm. Deze vorm wordt weergegeven als data zonder het uit te voeren in de browser. (voorbeelden: Convert & to & Convert < to < escape all characters with the \uXXXX unicode escaping format, bij JavaScript)
- Input validation
  Zorg ervoor dat alle input die binnen komt in de applicatie wordt 'gezuiverd' en gevalideerd.
- Nieuwe technieken zoals Content Security Policy (CSP)
  CSP ondersteunt een standaard HTTP header die toelaat om sources van toegelaten/vertrouwde content die mag geladen worden op hun website te laten declareren door de website owners.

18.5 Insecure direct object references

Problemen hierbij zijn dat er geen restricties (of te weinig) zijn op objecten waar de gebruiker geen recht op heeft, door het gebruik van *presentation layer access control* en door verborgen velden te gebruiken om object referenties op te slaan. Bij het laatste probleem kan je de object referenties gemakkelijk zien in de *source-code*.

De *attacker* zal proberen om data te wijzigen om ongeautoriseerde objecten te kunnen raadplegen. En zal de verborgen velden zichtbaar maken.

Oplossingen:

- Zo weinig mogelijk referenties zenden
- Gebruik een indirect reference map[6]
- Probeer access deftig te checken voor elke gevoelige acties.

De attacker kan bijvoorbeeld proberen URL's te wijzigen om belangrijke objecten te kunnen bemachtigen.

- Wijzigen file:
  `.../ImageServlet?url=http://backendhost/images/bg.gif`
  → `.../ImageServlet?url=file:////etc/passwd`
- Wijzigen accountnummer:
  `http://www.mybank.com/user?acct=6065` → `http://www.mybank.com/user?acct=6066`

18.6 Security misconfiguration

Bij configuratie van servers, platformen, applicaties, *frameworks*, ... kunnen er problemen zijn door de *default security settings*, *default* paswoorden te gebruiken, of onvoldoende te wijzigen, alsook het niet regelmatig updaten van *security patches*.

Dit kunnen we oplossen door regelmatig upgrades uit te voeren en een deftige configuratie uit te voeren. Het kan ook nuttig zijn om regelmatig te scannen. Alsook het vermijden van nodige prop's draaien op servers, OS strippen, ...

[6] Dit is een mapping tussen doorgestuurde referentie en effectieve referentie. Hierdoor zien de gebruikers een referentie, maar de referentie naar de DB zal worden *gemapped*. Zo zal er dus nooit interne referenties worden doorgegeven.

18.7 Sensitive data exposure

Onvoldoende beveiliging van gevoelige data door slecht beveiligde (onbeveiligde) opslag, *clear text transmission*, zwakke encryptie, slecht web beveiliging.

Oplossingen:

- Geen gevoelige data opslaan als het niet nodig is
- Als het nodig is encrypteer en hash paswoorden
- *Autocomplete* van forms halen

18.8 Missing function level access control

Tijdens het debuggen kan het zijn dat de programmeur een functie invoert om het debug-proces te vergemakkelijken. Als de programmeur deze vergeet weg te halen tijdens de release kan een attacker daar gebruik van maken. De programmeur kan ook een slechte vorm van autorisatie hebben ontwikkeld doorheen de applicatie of de programmeur heeft ze niet voldoende toegepast doorheen de applicatie, waardoor er delen slecht geautoriseerd zijn.

Oplossing:
Implementeer een consistente autorisatie doorheen de applicatie (container-based of code-based)

18.9 Cross site request forgery

Bij *cross site request forgery* gaat de attacker, de gebruiker ongewilde operaties laten uitvoeren binnen een applicatie. Voorbeeld: Attacker stuurt een mail met een link naar gebruiker, de gebruiker stuurt een request (via de link) naar de webapplicatie, waarbij ze al aangemeld (geautoriseerd) is. De link bevat het volgende: `http link + <script> parameters steken in body </script>`. Deze vorm van XSS kan dus misbruikt worden als de websites onvoldoende acties autoriseren. Oplossing:
Gebruik maken van een CSRF token, deze vraagt een bevestiging via een confirmationID.

18.10 Using components with known vulnerabilities

Het kan zijn dat je gebruik maakt van software die *libraries* of componenten gebruiken die gekende *vulnerabilities* hebben.

De oplossingen hiervoor zijn *straight-forward* en simpel, check regelmatig CVE voor gekende *vulnerabilities*, controleer de gebruikte frameworks/Services die worden gebruikt.

18.11 Unvalidated redirects and forwards

De attacker kan de gebruiker laten directen na het invullen van een form. Hierdoor kan de gebruiker geleid worden naar een *malicious* site, kan de *authentication* of autorisatie validatie worden overgeslagen.

Oplossingen:

- Vermijd redirects/forwards
- Vermijd redirection gebaseerd op de user input
- Als het dan toch moet, valideer en limiteer de *scope*

18.12 Business logic attacks

De attacker kan het gedrag van de applicatie misbruiken om *business damage* te veroorzaken. Dit door het misbruiken van functionaliteiten, misbruiken van opslagcapaciteiten, DOS, inhoud *spoofing*, ... Voorbeeld: In een cinema kon je de beschikbare plaats selecteren. Als men toen de transactie sloot werd deze plaats niet terug vrijgegeven. Zo kan men alle plaatsen afgaan en telkens de transactie afsluiten en zo de cinema-zaal voor zich alleen hebben.

18.13 Secure development lifecycles

Om nog veiligere software te ontwikkelen maakt men gebruik van een *software security improvement program*. Dit is een model die een ontwikkelaar best volgt (net zoals bij sectie 1 op pagina 4).

19 Bespreek de eigendomsrechten i.v.m. software

Europese richtlijn: EG-richtlijn 2001/29/EG over het auteursrecht in de informatiemaatschappij. Enkel op de code (of iets dat in code kan worden omgezet) kan je auteursrechten krijgen, niet de functionaliteit. Het idee behoort dus niet tot de auteursrechten, het moet niet nieuw zijn, een eigen schepping schendt niet de auteursrechten. De werkgever is eigenaar tenzij anders vermeld in een contract.

Software mag gereproduceerd en aangepast worden als het noodzakelijk is voor gebruik (debug, backup kopie, familie, collega's ˜licentieovereenkomst). De EG-richtlijn zegt dat er een wisselwerking met andere programma's mogelijk moet gemaakt worden, om monopolie tegen te gaan. Het analyseren van software mag, maar het decompileren niet (tenzij om compabiliteit te verkrijgen).

19.1 Databanken

De inhoud van de databank en de databank zelf kan worden beschermd.

Inhoud van databank De inhoud van een databank kan worden beschermd als er originele teksten, foto's, ... instaan. Het is echter niet beschermd als het feitenmateriaal bevat zoals adresgegevens, beurskoersen, vluchtgegevens, ... of officiële akten, ... Bij een verzameling van feitelijke gegevens zijn we wel tijdelijk beschermd voor 15 jaar (elke keer als er een belangrijke investering wordt gedaan). Opgelet dit geldt enkel binnen de EG.

Databank zelf De databank zelf is enkel beschermd als er een creatieve selectie is van materiaal, als er een originele ordening/structuur wordt gebruikt.

19.2 Privacy

Als we persoonsgegevens verwerken dan moeten we enkel gegevens opslaan/vragen die echt nodig zijn. Het ras, politieke overtuiging, gezondheid, veroordelingen, ... mogen niet gevraagd/opgeslagen worden tenzij ze hun intentie kunnen waarborgen. Er moeten veiligheidsmaatregelen worden voorzien tegen vernietiging, ongewenste toegang, ... Iedereen moet weten welke gegevens er over hen worden bijgehouden en waarom deze gegevens zijn opgeslagen in hun databanken.

19.3 Licenties

Een software-licentie is een rechtmatig instrument (contract) i.v.m. gebruik en distributie van software. Software is copyright-protected (auteursrecht) behalve als de software public domain materiaal is. Typisch geeft men de eindgebruiker het recht om één of meerdere kopies te gebruiken.

De licenties kennen rechten toe en leggen beperkingen op i.v.m. het gebruik. De licenties verduidelijking de aansprakelijkheden (liability), garanties (warranties), het afstand doen van garantie (warranty disclaimers), het vrijwaring/schadeloosstelling (indemnity) als software intellectual property rights van anderen schendt. Categoriën zijn *proprietary* en *free en open source*.

19.3.1 Public domain software

Bij *public domain software* wordt software expliciet in publiek domein geplaatst, de auteur doet expliciet afstand van zijn rechten.

19.3.2 Free software (freeware)

Public domain wordt soms verkeerd gebruikt voor software met free software licentie. Freeware bevat belangrijke hoeveelheid rechten (bv recht om software aan te passen en verder te verdelen) maar sommige rechten blijven bij auteur (bv copyleft (originele rechten moeten blijven gelden), niet commerciëel gebruiken).

19.3.3 GPL – GNU General Public License

De filosofie achter GPL is dat gebruikers software mogen runnen, het programma bestuderen, wijzigen of herverdelen, en deze gewijzigde versies verdelen. De software is en blijft vrij ook als iemand anders het wijzigt en/of verdeelt, dit is *copyleft*. De software is copyrighted, maar in plaats van het gebruik te beperken zoals *proprietary software*, verzekert het dat elke gebruiker de vrijheid heeft en zal hebben.

GNU GPL beperkt mensen niet in wat ze met de software doen, maar stopt hen van het opleggen van beperkingen aan anderen.